

Enterprise Java (Spring/Spring Boot) Course Notes

jim stafford

Fall 2024 2022-08-06: Built: 2024-12-03 22:36 EST

Table of Contents

1. Abstract	1
Enterprise Computing with Java (605.784.8VL) Course Syllabus	2
2. Course Description	3
2.1. Meeting Times/Location	3
2.2. Course Goal	3
2.3. Description	4
2.4. Student Background	4
2.5. Student Commitment	4
2.6. Course Text(s)	5
2.7. Required Software	5
2.8. Course Structure	5
2.9. Grading	6
2.10. Grading Policy	7
2.11. Academic Integrity	7
2.12. Instructor Availability	7
2.13. Communication Policy	7
2.14. Office Hours	8
3. Course Assignments	9
3.1. General Requirements	9
3.2. Submission Guidelines	9
4. Syllabus	11
Development Environment Setup	14
5. Introduction	15
5.1. Goals	15
5.2. Objectives	15
6. Software Setup	16
6.1. Java JDK (immediately)	16
6.2. Git Client (immediately)	17
6.3. Maven 3 (immediately)	18
6.4. Java IDE (immediately)	20
6.5. Web API Client tool (not immediately)	20
6.6. Optionally Install Docker (not immediately)	21
6.7. MongoDB (later)	23
Introduction to Enterprise Java Frameworks	24
7. Introduction	25
7.1. Goals	25
7.2. Objectives	25
8. Code Reuse	26

8.1. Code Reuse Trade-offs	26
8.2. Code Reuse Constructs	26
8.3. Code Reuse Styles	26
9. Frameworks	28
9.1. Framework Informal Description	28
9.2. Framework Characteristics	28
10. Framework Enablers	30
10.1. Dependency Injection	30
10.2. POJO	30
10.3. Component	30
10.4. Bean	30
10.5. Container	31
10.6. Interpose	31
11. Language Impact on Frameworks	34
11.1. XML Configurations	34
11.2. Annotations	34
11.3. Lambdas	35
12. Key Frameworks	36
12.1. CGI Scripts	36
12.2. JavaEE	36
12.3. Spring	36
12.4. Jakarta Persistence API (JPA)	37
12.5. Spring Data	37
12.6. Spring Boot	38
13. Summary	39
Pure Java Main Application	40
14. Introduction	41
14.1. Goals	41
14.2. Objectives	41
15. Simple Java Class with a Main	42
16. Project Source Tree	43
17. Building the Java Archive (JAR) with Maven	44
17.1. Add Core pom.xml Document	44
17.2. Add Optional Elements to pom.xml	44
17.3. Define Plugin Versions	45
17.4. pluginManagement vs. plugins	45
18. Build the Module	47
19. Project Build Tree	49
20. Resulting Java Archive (JAR)	50
21. Execute the Application	51
22. Configure Application as an Executable JAR	52

22.1. Add Main-Class property to MANIFEST.MF	52
22.2. Automate Additions to MANIFEST.MF using Maven	52
23. Execute the JAR versus just adding to classpath	53
24. Configure pom.xml to Test	54
24.1. Execute JAR as part of the build	55
25. Summary	56
Simple Spring Boot Application	57
26. Introduction	58
26.1. Goals	58
26.2. Objectives	58
27. Spring Boot Maven Dependencies	59
28. Parent POM	60
28.1. Define Version for Spring Boot artifacts	60
28.2. Import springboot-dependencies-plugin	61
29. Local Child/Leaf Module POM	62
29.1. Declare pom inheritance in the child pom.xml	62
29.2. Declare dependency on artifacts used	63
30. Simple Spring Boot Application Java Class	65
30.1. Module Source Tree	65
30.2. @SpringBootApplication Aggregate Annotation	66
31. Spring Boot Executable JAR	67
31.1. Building the Spring Boot Executable JAR	67
31.2. Java MANIFEST.MF properties	68
31.3. JAR size	69
31.4. JAR Contents	69
31.5. Execute Command Line	70
32. Add a Component to Output Message and Args	72
32.1. @Component Annotation	72
32.2. Interface: CommandLineRunner	72
32.3. @ComponentScan Tree	73
33. Running the Spring Boot Application	74
33.1. Implementation Note	74
34. Configure pom.xml to Test	75
34.1. Execute JAR as part of the build	75
35. Summary	77
Assignment 0: Application Build	78
36. Part A: Build Pure Java Application JAR	80
36.1. Purpose	80
36.2. Overview	80
36.3. Requirements	81
36.4. Grading	81

36.5. Additional Details	81
37. Part B: Build Spring Boot Executable JAR	83
37.1. Purpose	83
37.2. Overview	83
37.3. Requirements	84
37.4. Grading	84
37.5. Additional Details	84
Bean Factory and Dependency Injection	86
38. Introduction	87
38.1. Goals	87
38.2. Objectives	87
39. Hello Service	88
39.1. Hello Service API	88
39.2. Hello Service StdOut	88
39.3. Hello Service API pom.xml	89
39.4. Hello Service StdOut pom.xml	89
39.5. Hello Service Interface	90
39.6. Hello Service Sample Implementation	90
39.7. Hello Service Modules Complete	91
39.8. Hello Service API Maven Build	91
39.9. Hello Service StdOut Maven Build	92
40. Application Module	94
40.1. Application Maven Dependency	94
40.2. Viewing Dependencies	95
40.3. Application Java Dependency	95
41. Dependency Injection	97
42. Spring Dependency Injection	98
42.1. @Autowired Annotation	98
42.2. Dependency Injection Flow	99
43. Bean Missing	100
43.1. Bean Missing Error Solution(s)	100
44. @Configuration classes	101
45. @Bean Factory Method	102
46. @Bean Factory Used	103
47. Factory Alternative: XML Configuration	104
48. @Configuration Alternatives	106
48.1. Example Lifecycle POJO	106
48.2. Example Lifecycle @Configuration Class	107
48.3. Consuming @Bean Factories within Same Class	107
48.4. Using Defaults (Singleton, Proxy)	107
48.5. Prototype Scope, Proxy True	108

48.6. Prototype Scope, Proxy False	109
48.7. Singleton Scope, Proxy False	109
48.8. @Configuration Takeaways	110
49. Summary	111
Value Injection	112
50. Introduction	113
50.1. Goals	113
50.2. Objectives	113
51. @Value Annotation	114
51.1. Value Not Found	114
51.2. Value Property Provided by Command Line	115
51.3. Default Value	115
52. Constructor Injection	116
52.1. Constructor Injection Solution	116
53. @PostConstruct	117
54. Property Types	118
54.1. non-String Property Types	118
54.2. Collection Property Types	118
54.3. Custom Delimiters (using Spring SpEL)	119
54.4. Map Property Types	119
54.5. Map Element	120
54.6. System Properties	120
54.7. Property Conversion Errors	121
55. Summary	122
Property Source	123
56. Introduction	124
56.1. Goals	124
56.2. Objectives	124
57. Property File Source(s)	125
57.1. Property File Source Example	125
57.2. Example Property File Contents	126
57.3. Non-existent Path	127
57.4. Path not Ending with Slash ("/")	128
57.5. Alternate File Examples	128
57.6. Series of files	129
58. @PropertySource Annotation	130
59. Profiles	132
59.1. Default Profile	133
59.2. Specific Active Profile	134
59.3. Multiple Active Profiles	134
59.4. No Associated Profile	135

60. Property Placeholders	136
60.1. Placeholder Demonstration	136
60.2. Placeholder Property Files	137
60.3. Placeholder Value Defined Internally	137
60.4. Placeholder Value Defined in Profile	137
60.5. Multiple Active Profiles	138
60.6. Mixing Names, Profiles, and Location	138
61. Other Common Property Sources	140
61.1. Property Source	140
61.2. application.properties	141
61.3. Profiles	141
61.4. Environment Variables	141
61.5. Java System Properties	142
61.6. spring.application.json	143
61.7. Command Line Arguments	143
61.8. @SpringBootTest.properties	144
62. Summary	145
Configuration Properties	146
63. Introduction	147
63.1. Goals	147
63.2. Objectives	147
64. Mapping properties to @ConfigurationProperties class	148
64.1. Mapped Java Class	148
64.2. Injection Point	149
64.3. Initial Error	149
64.4. Registering the @ConfigurationProperties class	150
64.5. Result	152
65. Metadata	153
65.1. Spring Configuration Metadata	153
65.2. Spring Configuration Processor	153
65.3. Javadoc Supported	154
65.4. Rebuild Module	154
65.5. IDE Property Help	155
66. Constructor Binding	156
66.1. Property Names Bound to Constructor Parameter Names	157
66.2. Constructor Parameter Name Mismatch	157
67. Validation	159
67.1. Validation Annotations	159
67.2. Validation Error	160
68. Boilerplate JavaBean Methods	161
68.1. Generating Boilerplate Methods with Lombok	161

68.2. Visible Generated Constructs	162
68.3. Lombok Build Dependency	163
68.4. Example Output	163
69. Relaxed Binding	165
69.1. Relaxed Binding Example JavaBean	165
69.2. Relaxed Binding Example Properties	165
69.3. Relaxed Binding Example Output	166
70. Nested Properties	167
70.1. Nested Properties JavaBean Mapping	167
70.2. Nested Properties Host JavaBean Mapping	167
70.3. Nested Properties Output	168
71. Property Arrays	169
71.1. Property Arrays Definition	169
71.2. Property Arrays Output	170
72. System Properties	171
72.1. System Properties Usage	171
73. @ConfigurationProperties Class Reuse	173
73.1. @ConfigurationProperties Class Reuse Mapping	173
73.2. @ConfigurationProperties @Bean Factory	174
73.3. Injecting ownerProps	174
73.4. Injection Matching	175
73.5. Ambiguous Injection	175
73.6. Injection @Qualifier	176
73.7. way1: Create Custom @Qualifier Annotation	176
73.8. way2: @Bean Factory Method Name as Qualifier	177
73.9. way3: Match @Bean Factory Method Name	177
73.10. Ambiguous Injection Summary	178
74. Summary	179
Auto Configuration	180
75. Introduction	181
75.1. Goals	181
75.2. Objectives	181
76. Review: Configuration Class	183
76.1. Separate @Configuration Class	183
77. Conditional Configuration	184
77.1. Property Value Condition Satisfied	184
77.2. Property Value Condition Not Satisfied	185
78. Two Primary Configuration Phases	186
79. Auto-Configuration	187
79.1. @AutoConfiguration Annotation	188
79.2. Supporting @ConfigurationProperties	188

79.3. Locating Auto Configuration Classes	188
79.4. META-INF Auto-configuration Metadata File	189
79.5. Spring Boot 2 META-INF/spring.factories	189
79.6. Example Auto-Configuration Module Source Tree	190
79.7. Auto-Configuration / Starter Roles/Relationships	190
79.8. Example Starter Module pom.xml	191
79.9. Application Starter Dependency	192
79.10. Starter Brings in Pertinent Dependencies	192
80. Configured Application	193
80.1. Review: Unconditional Auto-Configuration Class	193
80.2. Review: Starter Module Default	193
80.3. Produced Default Starter Greeting	194
80.4. User-Application Supplies Property Details	194
81. Auto-Configuration Conflict	196
81.1. Review: Conditional @Bean Factory	196
81.2. Potential Conflict	196
81.3. @ConditionalOnMissingBean	197
81.4. Bean Conditional Example Output	197
82. Resource Conditional and Ordering	199
82.1. @AutoConfiguration Alternative	199
82.2. Registering Second Auto-Configuration Class	199
82.3. Resource Conditional Example Output	200
83. @Primary	201
83.1. @Primary Example Output	202
84. Class Conditions	203
84.1. Class Conditional Example	203
85. Excluding Auto Configurations	204
86. Debugging Auto Configurations	205
86.1. Conditions Evaluation Report	205
86.2. Conditions Evaluation Report Example	205
86.3. Condition Evaluation Report Results	206
86.4. Actuator Conditions	206
86.5. Activating Actuator Conditions	206
86.6. Actuator Environment	208
86.7. Actuator Links	208
86.8. Actuator Environment Report	208
86.9. Actuator Specific Property Source	209
86.10. More Actuator	209
87. Summary	210
Logging	211
88. Introduction	212

88.1. Why log?	212
88.2. Why use a Logger over System.out?	212
88.3. Goals	212
88.4. Objectives	213
89. Starting References	214
90. Logging Dependencies	215
90.1. Logging Libraries	215
90.2. Spring and Spring Boot Internal Logging	216
91. Getting Started	217
91.1. System.out	217
91.2. System.out Output	217
91.3. Turning Off Spring Boot Logging	217
91.4. Getting a Logger	218
91.5. Java Util Logger Interface Example	218
91.6. JUL Example Output	219
91.7. SLF4J Logger Interface Example	220
91.8. SLF4J Example Output	221
91.9. Lombok SLF4J Declaration Example	221
91.10. Lombok Example Output	221
91.11. Lombok Dependency	222
92. Logging Levels	223
92.1. Common Level Use	223
92.2. Log Level Adjustments	224
92.3. Logging Level Example Calls	224
92.4. Logging Level Output: INFO	224
92.5. Logging Level Output: DEBUG	225
92.6. Logging Level Output: TRACE	226
92.7. Logging Level Output: WARN	226
92.8. Logging Level Output: OFF	227
93. Discarded Message Expense	228
93.1. Blind String Concatenation	228
93.2. Verbosity Check	228
93.3. SLF4J Parameterized Logging	229
93.4. Simple Performance Results: Disabled	230
93.5. Simple Performance Results: Enabled	231
94. Exception Logging	232
94.1. Exception Example Output	232
94.2. Exception Logging and Formatting	233
95. Logging Pattern	234
95.1. Default Console Pattern	234
95.2. Default Console Pattern Output	235

95.3. Variable Substitution	236
95.4. Conditional Variable Substitution	236
95.5. Date Format Pattern	236
95.6. Log Level Pattern	237
95.7. Conversion Pattern Specifiers	237
95.8. Format Modifier Impact Example	238
95.9. Example Override	238
95.10. Expensive Conversion Words	239
95.11. Example Override Output	239
95.12. Layout Fields	240
96. Loggers	241
96.1. Logger Tree	241
96.2. Logger Inheritance	241
96.3. Logger Threshold Level Inheritance	242
96.4. Logger Effective Threshold Level Inheritance	242
96.5. Example Logger Threshold Level Properties	242
96.6. Example Logger Threshold Level Output	243
97. Appenders	244
97.1. Logger has N Appenders	244
97.2. Logger Configuration Files	244
97.3. Logback Root Configuration Element	245
97.4. Retain Spring Boot Defaults	245
97.5. Appender Configuration	245
97.6. Appenders Attached to Loggers	246
97.7. Appender Tree Inheritance	247
97.8. Appender Additivity Result	247
97.9. Logger Inheritance Tree Output	247
98. Mapped Diagnostic Context	249
98.1. MDC Example	249
98.2. MDC Example Pattern	250
98.3. MDC Example Output	250
98.4. Clearing MDC Context	251
99. Markers	252
99.1. Marker Class	252
99.2. Marker Example	252
99.3. Marker Appender Filter Example	253
99.4. Marker Example Result	254
100. File Logging	255
100.1. root Logger Appenders	255
100.2. FILE Appender Output	255
100.3. Spring Boot FILE Appender Definition	256

100.4. RollingFileAppender	257
100.5. SizeAndTimeBasedRollingPolicy	257
100.6. FILE Appender Properties	257
100.7. logging.file.path	258
100.8. logging.file.name	258
100.9. logging.logback.rollingpolicy.max-file-size Trigger	259
100.10. logging.pattern.rolling-file-name	260
100.11. Timestamp Rollover Example	260
100.12. History Compression Example	261
100.13. logging.logback.rollingpolicy.max-history Example	261
100.14. logging.logback.rollingpolicy.total-size-cap Index Example	262
100.15. logging.logback.rollingpolicy.total-size-cap no Index Example	262
101. Custom Configurations	264
101.1. Logback Configuration Customization	264
101.2. Provided Logback Includes	264
101.3. Customization Example: Turn off Console Logging	264
101.4. LOG_FILE Property Definition	265
101.5. Customization Example: Leverage Restored Defaults	266
101.6. Customization Example: Provide Override	266
102. Spring Profiles	267
103. Summary	268
Testing	269
104. Introduction	270
104.1. Why Do We Test?	270
104.2. What are Test Levels?	270
104.3. What are some Approaches to Testing?	270
104.4. Goals	270
104.5. Objectives	271
105. Test Constructs	272
105.1. Automated Test Terminology	272
105.2. Maven Test Types	272
105.3. Test Naming Conventions	273
105.4. Lecture Test Naming Conventions	273
106. Spring Boot Starter Test Frameworks	274
106.1. Spring Boot Starter Transitive Dependencies	274
106.2. Transitive Dependency Test Tools	275
107. JUnit Background	276
107.1. JUnit 5 Evolution	277
107.2. JUnit 5 Areas	278
107.3. JUnit 5 Module JARs	278
108. Syntax Basics	280

109. JUnit Vintage Basics	281
109.1. JUnit Vintage Example Lifecycle Methods.....	281
109.2. JUnit Vintage Example Test Methods	282
109.3. JUnit Vintage Basic Syntax Example Output.....	282
110. JUnit Jupiter Basics	284
110.1. JUnit Jupiter Example Lifecycle Methods	284
110.2. JUnit Jupiter Example Test Methods	285
110.3. JUnit Jupiter Basic Syntax Example Output	285
111. JUnit Jupiter Test Case Adjustments	287
111.1. Test Instance.....	287
111.2. Shared Instance State - PER_CLASS.....	287
112. Assertion Basics	290
112.1. Assertion Libraries	290
112.2. Example Library Assertions	292
112.3. Assertion Failures	293
112.4. Testing Multiple Assertions.....	294
112.5. Asserting Exceptions.....	295
112.6. Asserting Dates.....	297
113. Mockito Basics	299
113.1. Test Doubles	299
113.2. Mock Support	299
113.3. Mockito Learning Example Declarations.....	299
113.4. Mockito Learning Example Test	300
114. BDD Acceptance Test Terminology	302
114.1. Alternate BDD Syntax Support.....	302
114.2. Example BDD Syntax Support	302
114.3. Example BDD Syntax Output	303
114.4. JUnit Options Expressed in Properties	304
115. Tipping Example	305
116. Review: Unit Test Basics	306
116.1. Review: POJO Unit Test Setup	306
116.2. Review: POJO Unit Test	306
116.3. Review: Mocked Unit Test Setup	307
116.4. Review: Mocked Unit Test	307
116.5. @InjectMocks	308
117. Spring Boot Unit Integration Test Basics	309
117.1. Adding Spring Boot to Testing	309
117.2. @SpringBootTest	309
117.3. Default @SpringBootConfiguration Class	310
117.4. Conditional Components	310
117.5. Explicit Reference to @SpringBootConfiguration	310

117.6. Explicit Reference to Components	311
117.7. Active Profiles	311
117.8. Example @SpringBootTest Unit Integration Test	312
117.9. Example @SpringBootTest NTest Output	312
117.10. Alternative Test Slices	313
118. Mocking Spring Boot Unit Integration Tests	314
118.1. Example @SpringBoot/Mockito Test	314
119. Maven Unit Testing Basics	316
119.1. Maven Surefire Plugin	316
119.2. Filtering Tests	317
119.3. Filtering Tests Executed	317
119.4. Maven Failsafe Plugin	318
119.5. Failsafe Overhead	318
120. @TestConfiguration	320
120.1. Example Spring Context	320
120.2. Test TippingCalculator	320
120.3. Enable Component Replacement	321
120.4. Embedded TestConfiguration	322
120.5. External TestConfiguration	322
120.6. Using External Configuration	322
120.7. TestConfiguration Result	323
121. Summary	324
AutoRentals Assignment 1	325
122. Assignment 1a: App Config	327
122.1. @Bean Factory Configuration	327
122.2. Property Source Configuration	329
122.3. Configuration Properties	337
122.4. Auto-Configuration	343
123. Assignment 1b: Logging	351
123.1. Application Logging	351
123.2. Logging Efficiency	355
123.3. Appenders and Custom Log Patterns	356
124. Assignment 1c: Testing	361
124.1. Demo	361
124.2. Unit Testing	362
124.3. Mocks	364
124.4. Mocked Unit Integration Test	367
124.5. Unmocked/BDD Unit Integration Testing	369
HTTP API	371
125. Introduction	372
125.1. Goals	372

125.2. Objectives	372
126. World Wide Web (WWW)	373
126.1. Example WWW Information System	373
127. REST	374
127.1. HATEOAS	374
127.2. Clients Dynamically Discover State	374
127.3. Static Interface Contracts	375
127.4. Internet Scale	375
127.5. How RESTful?	375
127.6. Buzzword Association	376
127.7. REST-like or HTTP-based	376
127.8. Richardson MaturityModel (RMM)	376
127.9. "REST-like"/"HTTP-based" APIs	377
127.10. Uncommon REST Features Adopted	377
128. RMM Level 2 APIs	378
129. HTTP Protocol Embraced	379
130. Resource	380
130.1. Nested Resources	380
131. Uniform Resource Identifiers (URIs)	381
131.1. Related URI Terms	381
131.2. URI Generic Syntax	382
131.3. URI Component Examples	382
131.4. URI Characters and Delimiters	383
131.5. URI Percent Encoding	383
131.6. URI Case Sensitivity	384
131.7. URI Reference	384
131.8. URI Reference Terms	384
131.9. URI Naming Conventions	385
131.10. URI Variables	386
132. Methods	387
132.1. Additional HTTP Methods	387
133. Method Safety	388
133.1. Safe and Unsafe Methods	388
133.2. Violating Method Safety	388
134. Idempotent	390
134.1. Idempotent and non-Idempotent Methods	390
135. Response Status Codes	391
135.1. Common Response Status Codes	391
136. Representations	392
136.1. Content Type Headers	392
137. Links	394

138. Summary	395
Spring MVC	396
139. Introduction	397
139.1. Goals	397
139.2. Objectives	397
140. Spring Web APIs	398
140.1. Lecture/Course Focus	398
140.2. Spring MVC	398
140.3. Spring WebFlux	399
140.4. Synchronous vs. Asynchronous	399
140.5. Mixing Approaches	400
140.6. Choosing Approaches	401
141. Maven Dependencies	402
142. Sample Application	403
143. Annotated Controllers	404
143.1. Class Mappings	404
143.2. Method Request Mappings	405
143.3. Default Method Response Mappings	405
143.4. Executing Sample Endpoint	406
144. RestTemplate Client	408
144.1. JUnit Integration Test Setup	408
144.2. Form Endpoint URL	410
144.3. Obtain RestTemplate	410
144.4. Invoke HTTP Call	411
144.5. Evaluate Response	411
145. Spring Rest Clients	412
146. RestClient Client	413
146.1. Obtain RestClient	413
146.2. Invoke HTTP Call	413
147. WebClient Client	415
147.1. Obtain WebClient	415
147.2. Invoke HTTP Call	415
148. Spring HTTP Interface	417
149. Implementing Parameters	418
149.1. Controller Parameter Handling	418
149.2. Client-side Parameter Handling	419
149.3. Spring HTTP Interface Parameter Handling	420
150. Accessing HTTP Responses	421
150.1. Obtaining ResponseEntity	421
150.2. ResponseEntity<T>	422
151. Client Error Handling	423

151.1. RestTemplate Response Exceptions	423
151.2. RestClient Response Exceptions	424
151.3. WebClient Response Exceptions	425
151.4. Spring HTTP Interface Exceptions	426
151.5. Client Exceptions	426
152. Controller Responses	428
152.1. Controller Return ResponseEntity	428
152.2. Example ResponseEntity Responses	429
152.3. Controller Exception Handler	429
152.4. Simplified Controller Using ExceptionHandler	430
153. Summary	432
Controller/Service Interface	433
154. Introduction	434
154.1. Goals	434
154.2. Objectives	434
155. Roles	436
156. Error Reporting	437
156.1. Complex Object Result	437
156.2. Thrown Exception	437
156.3. Exceptions	438
156.4. Checked or Unchecked?	439
156.5. Candidate Client Exceptions	440
156.6. Service Errors	441
157. Controller Exception Advice	443
157.1. Service Method with Exception Logic	443
157.2. Controller Advice Class	444
157.3. Advice Exception Handlers	445
158. Summary	446
API Data Formats	447
159. Introduction	448
159.1. Goals	448
159.2. Objectives	448
160. Pattern Data Transfer Object	449
160.1. DTO Pattern Problem Space	449
160.2. DTO Pattern Solution Space	449
160.3. DTO Pattern Players	450
161. Sample DTO Class	451
162. Time/Date Detour	452
162.1. Pre Java 8 Time	452
162.2. java.time	452
162.3. Date/Time Formatting	453

162.4. Date/Time Exchange	454
163. Java Marshallers	456
164. JSON Content	457
164.1. Jackson JSON	457
164.2. JSON-B	461
165. XML Content	464
165.1. Jackson XML	465
165.2. JAXB	466
166. Configure Server-side Jackson	471
166.1. Dependencies	471
166.2. Configure ObjectMapper	471
166.3. Controller Properties	472
167. Client Marshall Request Content	475
168. Client Filters	477
168.1. RestTemplate and RestClient	477
168.2. WebClient	478
169. Date/Time Lenient Parsing and Formatting	480
169.1. Out of the Box Time-related Formatting	480
169.2. Out of the Box Time-related Parsing	481
169.3. JSON-B DATE_FORMAT Option	482
169.4. JSON-B Custom Serializer Option	482
169.5. Jackson Lenient Parser	483
170. Summary	485
Swagger	486
171. Introduction	487
171.1. Goals	487
171.2. Objectives	487
172. Swagger Landscape	488
172.1. Open API Standard	488
172.2. Swagger-based Tools	488
172.3. Springfox	489
172.4. Springdoc	489
173. Minimal Configuration	491
173.1. Springdoc Minimal Configuration	491
174. Example Use	493
174.1. Access Contest Controller POST Command	493
174.2. Invoke Contest Controller POST Command	493
174.3. View Contest Controller POST Command Results	494
175. Useful Configurations	495
175.1. Customizing Type Expressions	495
176. Client Generation	500

176.1. API Schema Definition	500
176.2. API Client Source Tree	500
176.3. OpenAPI Maven Plugin	501
176.4. OpenAPI Generated Target Tree	501
176.5. OpenAPI Compilation Dependencies	502
176.6. OpenAPI Client Build	503
177. Springdoc Summary	504
178. Summary	505
AutoRentals Assignment 2: API	506
179. Overview	507
179.1. Grading Emphasis	507
179.2. AutoRenter Support	507
180. Assignment 2a: Modules	512
180.1. Purpose	512
180.2. Overview	512
180.3. Requirements	513
180.4. Grading	514
180.5. Additional Details	515
181. Assignment 2b: Content	516
181.1. Purpose	516
181.2. Overview	516
181.3. Requirements	518
181.4. Grading	519
181.5. Additional Details	519
182. Assignment 2c: Resources	521
182.1. Purpose	521
182.2. Overview	521
182.3. Requirements	522
182.4. Grading	523
182.5. Additional Details	523
183. Assignment 2d: Client/API Interactions	525
183.1. Purpose	525
183.2. Overview	525
183.3. Requirements	526
183.4. Grading	527
183.5. Additional Details	527
184. Assignment 2e: Service/Controller Interface	529
184.1. Purpose	529
184.2. Overview	529
184.3. Requirements	531
184.4. Grading	533

184.5. Additional Details	534
185. Assignment 2f: Required Test Scenarios	535
185.1. Scenario: Creation of AutoRental.....	535
185.2. Scenario: Update of AutoRental.....	539
185.3. Requirements	542
185.4. Grading	542
185.5. Additional Details	542
Spring Security Introduction.....	543
186. Introduction	544
186.1. Goals	544
186.2. Objectives	544
187. Access Control	545
188. Privacy	546
188.1. Encoding	546
188.2. Encryption	546
188.3. Cryptographic Hash	547
189. Spring Web	549
190. No Security	550
190.1. Sample GET	550
190.2. Sample POST	550
190.3. Sample Static Content	551
191. Spring Security	553
191.1. Spring Core Authentication Framework	553
191.2. SecurityContext	555
192. Spring Boot Security AutoConfiguration	556
192.1. Maven Dependency	556
192.2. SecurityAutoConfiguration	557
192.3. WebSecurityConfiguration	558
192.4. UserDetailsServiceAutoConfiguration	558
192.5. SecurityFilterAutoConfiguration	558
193. Default FilterChain	559
194. Default Secured Application	560
194.1. Form Authentication Activated	560
194.2. Basic Authentication Activated	561
194.3. Authentication Required Activated	562
194.4. Username/Password can be Supplied	562
194.5. CSRF Protection Activated	563
194.6. Other Headers	563
195. Default FilterChainProxy Bean	565
196. Summary	570
Spring Security Authentication.....	571

197. Introduction	572
197.1. Goals	572
197.2. Objectives	572
198. Configuring Security	573
198.1. WebSecurityConfigurer and Component-based Approaches	573
198.2. Core Application Security Configuration	574
198.3. Ignoring Static Resources	576
198.4. SecurityFilterChain Matcher	578
198.5. SecurityFilterChain with Explicit MvcRequestMatcher	579
198.6. HttpSecurity Builder Methods	580
198.7. Match Requests	581
198.8. Authorize Requests	582
198.9. Authentication	583
198.10. Header Configuration	583
198.11. Stateless Session Configuration	584
199. Configuration Results	585
199.1. Successful Anonymous Call	585
199.2. Successful Authenticated Call	585
199.3. Rejected Unauthenticated Call Attempt	586
200. Authenticated User	587
200.1. Inject UserDetails into Call	587
200.2. Obtain SecurityContext from Holder	587
201. Swagger BASIC Auth Configuration	588
201.1. Swagger Authentication Configuration	588
201.2. Swagger Security Scheme	589
202. CORS	592
202.1. Default CORS Support	592
202.2. Browser and CORS Response	592
202.3. Enabling CORS	594
202.4. Constrained CORS	596
202.5. CORS Server Acceptance	596
202.6. CORS Server Rejection	597
202.7. Spring MVC @CrossOrigin Annotation	597
203. RestTemplate Authentication	599
203.1. ClientHttpRequestFactory	599
203.2. Anonymous RestTemplate	599
203.3. Authenticated RestTemplate	600
203.4. Authentication Integration Tests with RestTemplate	600
204. RestClient Authentication	601
204.1. Anonymous RestClient	601
204.2. Authenticated RestTemplate	601

205. Mock MVC Authentication	602
205.1. MockMvc Anonymous Call	602
205.2. MockMvc Authenticated Call	603
205.3. MockMvc does not require SpringBootTest	603
206. Summary	604
User Details	605
207. Introduction	606
207.1. Goals	606
207.2. Objectives	606
208. AuthenticationManager	607
208.1. ProviderManager	607
208.2. AuthenticationManagerBuilder	608
208.3. AuthenticationManagerBuilder Builder Methods	609
208.4. AuthenticationProvider	610
208.5. AbstractUserDetailsAuthenticationProvider	611
208.6. DaoAuthenticationProvider	611
208.7. UserDetailsServiceManager	612
209. AuthenticationManagerBuilder Configuration	613
209.1. Fully-Assembled AuthenticationManager	613
209.2. Directly Wire-up AuthenticationManager	614
209.3. Directly Wire-up Parent AuthenticationManager	614
209.4. Define Service and Encoder @Bean	615
209.5. Combine Approaches	618
210. UserDetailsService	621
211. PasswordEncoder	622
211.1. NoOpPasswordEncoder	622
211.2. BCryptPasswordEncoder	622
211.3. DelegatingPasswordEncoder	622
212. JDBC UserDetailsService	623
212.1. H2 Database	624
212.2. DataSource: Maven Dependencies	624
212.3. JDBC UserDetailsService	624
212.4. Autogenerated Database URL	625
212.5. Specified Database URL	625
212.6. Enable H2 Console Security Settings	625
212.7. Form Login	628
212.8. H2 Login	628
212.9. H2 Console	628
212.10. Create DB Schema Script	629
212.11. Schema Creation	630
212.12. Create User DB Populate Script	630

212.13. User DB Population	631
212.14. H2 User Access	631
212.15. Authenticate Access using JDBC UserDetailsService	632
212.16. Encrypting Passwords	632
213. Final Examples	634
213.1. Authenticate to All Three UserDetailsServices	634
213.2. Authenticate to All Three Users	634
214. Summary	635
Authorization	636
215. Introduction	637
215.1. Goals	637
215.2. Objectives	637
216. Authorities, Roles, Permissions	638
217. Authorization Constraint Types	639
217.1. Path-based Constraints	639
217.2. Annotation-based Constraints	641
218. Setup	643
218.1. Who Am I Controller	643
218.2. Demonstration Users	644
218.3. Core Security FilterChain Setup	644
218.4. Controller Operations	645
219. Path-based Authorizations	646
219.1. Path-based Role Authorization Constraints	646
219.2. Example Path-based Role Authorization (Sam)	647
219.3. Example Path-based Role Authorization (Woody)	647
220. Path-based Authority Permission Constraints	649
220.1. Path-based Authority Permission (Norm)	649
220.2. Path-based Authority Permission (Frasier)	650
220.3. Path-based Authority Permission (Sam and Woody)	650
220.4. Other Path Constraints	650
220.5. Other Path Constraints Usage	651
221. Authorization	653
221.1. Review: FilterSecurityInterceptor At End of Chain	653
221.2. Attempt Authorization Call	653
221.3. FilterSecurityInterceptor Calls	654
222. Role Inheritance	655
222.1. Role Inheritance Definition	655
223. @Secured	657
223.1. Enabling @Secured Annotations	657
223.2. @Secured Annotation	658
223.3. @Secured Annotation Checks	658

223.4. @Secured Many Roles	658
223.5. @Secured Now Processes Roles and Permissions	659
223.6. @Secured Role Inheritance	659
224. Controller Advice	661
224.1. AccessDeniedException Exception Handler	661
224.2. AccessDeniedException Exception Result	662
225. JSR-250	663
225.1. Enabling JSR-250	663
225.2. @RolesAllowed Annotation	663
225.3. @RolesAllowed Annotation Checks	664
225.4. Multiple Roles	664
225.5. Multiple Role Check	664
225.6. JSR-250 Does not Support Non-Role Authorities	665
225.7. JSR-250 Role Inheritance	665
226. Expressions	666
226.1. Expression Role Constraint	666
226.2. Expression Role Constraint Checks	667
226.3. Expressions Support Permissions and Role Inheritance	667
226.4. Supports Permissions and Boolean Logic	668
227. Summary	670
Enabling HTTPS	671
228. Introduction	672
228.1. Goals	672
228.2. Objectives	672
229. HTTP Access	673
230. HTTPS	674
230.1. HTTPS/TLS	674
230.2. Keystores	674
230.3. Tools	674
230.4. Self Signed Certificates	674
231. Enable HTTPS/TLS in Spring Boot	676
231.1. Generate Self-signed Certificate	676
231.2. Place Keystore in Reference-able Location	677
231.3. Add TLS properties	678
232. Untrusted Certificate Error	679
232.1. Option: Supply Trusted Certificates	679
232.2. Option: Accept Self-signed Certificates	680
233. Optional Redirect	682
233.1. HTTP:8080 ⇒ HTTPS:8443 Redirect Example	682
233.2. Follow Redirects	683
233.3. Caution About Redirects	684

234. Maven Unit Integration Test	685
234.1. JUnit @SpringBootTest	685
234.2. application-https.properties [REVIEW]	686
234.3. application-ntest.properties	686
234.4. ServerConfig	687
234.5. Maven Dependencies	687
234.6. HTTPS ClientHttpRequestFactory	687
234.7. SSL Factory	688
234.8. JUnit @Test	689
235. Summary	691
AutoRentals Assignment 3: Security	692
236. Assignment Starter	693
237. Assignment Support	696
237.1. High Level View	696
238. Assignment 3a: Security Authentication	699
238.1. Anonymous Access	699
238.2. Authenticated Access	702
238.3. User Details	706
239. Assignment 3b: Security Authorization	710
239.1. Authorities	710
239.2. Authorization	712
240. Assignment 3c: HTTPS	716
240.1. HTTPS	716
241. Assignment 3d: AOP and Method Proxies	718
241.1. Reflection	718
241.2. Dynamic Proxies	720
241.3. Aspects	724
Spring AOP and Method Proxies	729
242. Introduction	730
242.1. Goals	730
242.2. Objectives	730
243. Rationale	732
243.1. Adding More Cross-Cutting Capabilities	732
243.2. Using Proxies	732
244. Reflection	734
244.1. Reflection Method	734
244.2. Calling Reflection Method	735
244.3. Reflection Method Result	735
245. JDK Dynamic Proxies	737
245.1. Creating Dynamic Proxy	737
245.2. Generated Dynamic Proxy Class Output	738

245.3. Alternative Proxy All Construction	738
245.4. InvocationHandler Class	738
245.5. InvocationHandler invoke() Method	739
245.6. Calling Proxied Object	740
246. CGLIB	741
246.1. Creating CGLIB Proxy.....	741
246.2. MethodInterceptor Class	742
246.3. MethodInterceptor intercept() Method	742
246.4. Calling CGLIB Proxied Object.....	743
246.5. Dynamic Object CGLIB Proxy.....	743
247. AOP Proxy Factory	745
248. Interpose	748
249. Spring AOP	749
249.1. AOP Definitions	749
249.2. Enabling Spring AOP.....	750
249.3. Aspect Class.....	750
249.4. Pointcut	751
249.5. Pointcut Expression	751
249.6. Example Pointcut Definition.....	752
249.7. Combining Pointcut Expressions.....	752
249.8. Advice	753
250. Pointcut Expression Examples.....	754
250.1. execution Pointcut Expression.....	754
250.2. within Pointcut Expression.....	756
250.3. target and this Pointcut Expressions	756
251. Advice Parameters	757
251.1. Typed Advice Parameters	757
251.2. Multiple,Typed Advice Parameters	758
251.3. Annotation Parameters	758
251.4. Target and Proxy Parameters.....	759
251.5. Dynamic Parameters	760
251.6. Dynamic Parameters Output	760
252. Advice Types	761
252.1. @Before	761
252.2. @AfterReturning	762
252.3. @AfterThrowing	762
252.4. @After	763
252.5. @Around	763
253. Introductions	765
253.1. Component Introductions	765
253.2. Data Introductions	769

253.3. JSON Output Result	775
254. Other Features	777
255. Summary	778
Maven Integration Test	779
256. Introduction	780
256.1. Goals	780
256.2. Objectives	780
257. Maven Integration Test	781
257.1. Maven Integration Test Phases	781
258. Maven Integration Test Plugins	783
258.1. Spring Boot Maven Plugin	783
258.2. Build Helper Maven Plugin	786
258.3. Failsafe Plugin	787
259. Integration Test Client	790
259.1. JUnit @SpringBootTest	790
259.2. ClientTestConfiguration	790
259.3. username/password Credentials	791
259.4. ServerConfig	791
259.5. authnUrl URI	792
259.6. authUser RestTemplate	792
259.7. HTTP ClientHttpRequestFactory	793
259.8. JUnit @Test	793
260. IDE Execution	794
261. Maven Execution	795
261.1. Maven Verify	795
262. Maven Configuration Reuse	798
262.1. it Maven profile	798
262.2. Failsafe HTTPS/IT Profile Example	799
263. Summary	803
Docker Integration Testing	804
264. Introduction	805
264.1. Goals	805
264.2. Objectives	805
265. Disable Spring Boot Plugin Start/Stop	806
266. Docker Maven Plugin	807
266.1. Executions	807
266.2. Configuring with Properties	808
266.3. Configuration Properties	809
267. Concurrent Testing	811
268. Non-local Docker Host	812
268.1. Linux Work-around	814

268.2. Failsafe	814
268.3. Resolving docker.hostname	815
268.4. Example Output	817
269. Summary	818
Docker Compose	819
270. Introduction	820
270.1. Goals	820
270.2. Objectives	820
271. Development and Integration Testing with Real Resources	821
271.1. Managing Images	821
272. Docker Compose	823
272.1. Docker Compose is Local to One Machine	823
273. Docker Compose Configuration File	824
273.1. mongo Service Definition	824
273.2. postgres Service Definition	825
273.3. api Service Definition	826
273.4. Project Directory	827
273.5. Build/Download Images	828
273.6. Default Port Assignments	828
273.7. Compose Override Files	829
273.8. Compose Override File Naming	829
273.9. Multiple Compose Files	829
273.10. Environment Files	830
274. Docker Compose Commands	832
274.1. Build Source Images	832
274.2. Start Services in Foreground	832
274.3. Project Name	832
274.4. Start Services in Background	833
274.5. Access Service Logs	833
274.6. Stop Running Services	834
275. Docker Cleanup	835
275.1. Docker Image Prune	836
275.2. Docker System Prune	836
275.3. Image Repository State After Pruning	837
276. Summary	838
Docker Compose Integration Testing	839
277. Introduction	840
277.1. Goals	841
277.2. Objectives	841
278. Starting Point	842
278.1. Maven Dependencies	842

278.2. Unit Integration Test Setup	843
278.3. Unit Integration Test	843
278.4. DataSource Service Injection	844
278.5. DataSource Service Injection Test	845
278.6. DataSource Service Injection Test Result	845
279. Docker Compose Database Services	846
279.1. Docker Compose File	846
279.2. Manually Starting Database Containers	846
279.3. Postgres Container IT Test	847
279.4. Postgres Container IT Result	848
279.5. MongoDB Container IT Test	849
279.6. MongoClient Service Injection	850
279.7. MongoDB Container IT Result	851
280. Automating Container DB IT Tests	852
280.1. Generate Random Available Port Numbers	852
280.2. Launch Docker Compose with Variables	853
280.3. Filtering Test Resources	854
280.4. Failsafe	857
280.5. Test Execution	857
281. CI/CD Test Execution	860
282. Adding API Container	861
282.1. API in Docker Compose	861
282.2. API Dockerfile	861
282.3. run_env.sh	862
282.4. API Container Integration Test	865
283. Summary	869
Testcontainers Unit Integration Testing	870
284. Introduction	871
284.1. Goals	871
284.2. Objectives	871
285. Testcontainers Overview	872
286. Base Example	873
286.1. Maven Dependencies	873
286.2. Module Main Tree	873
286.3. Injected RestController Component	874
286.4. In-Memory Example	875
287. Postgres Container Example	878
287.1. Maven Dependencies	878
287.2. Unit Integration Test Setup	879
287.3. Inject Postgres Container DataSource into Test	883
287.4. Inject Postgres Container DataSource into Server-side	883

287.5. Runtime Docker Containers	884
288. MongoDB Container Example	885
288.1. Maven Dependencies	885
288.2. Unit Integration Test Setup	885
288.3. Inject MongoDB Container Client into Test	888
288.4. Inject MongoDB Container Client into Server-side	889
288.5. Runtime Docker Containers	889
289. Docker Compose Example	891
289.1. Maven Aspects	891
289.2. Example Files	892
289.3. Integration Test Setup	894
289.4. Inject Postgres Connection into Test	901
289.5. Inject Postgres Connection into Server-Side	902
289.6. Inject MongoClient into Test	902
289.7. Inject MongoClient into Server-side	903
290. Summary	905
AutoRentals Assignment 4: Integration Testing	906
291. Assignment 4a: Executable JAR Option	907
291.1. Purpose	907
291.2. Overview	907
291.3. Requirements	907
292. Assignment 4b: Docker Plugin Option	914
292.1. Docker Image	914
292.2. Docker Integration Test	917
293. Assignment 4c: Testcontainers Option	925
293.1. Docker Image	925
293.2. Testcontainers Integration Test	928
RDBMS	934
294. Introduction	935
294.1. Goals	935
294.2. Objectives	935
295. Schema Concepts	936
295.1. RDBMS Tables/Columns	936
295.2. Column Data	936
295.3. Column Types	936
295.4. Example Column Types	937
295.5. Constraints	937
295.6. Primary Key	938
295.7. UUID	939
295.8. Database Sequence	939
296. Example POJO	942

297. Schema	943
297.1. Schema Creation	943
297.2. Example Schema	943
298. Schema Command Line Population	945
298.1. Schema Result	945
298.2. List Tables	946
298.3. Describe Song Table	946
299. RDBMS Project	947
299.1. RDBMS Project Dependencies	947
299.2. RDBMS Access Objects	948
299.3. RDBMS Connection Properties	948
300. Schema Migration	950
300.1. Flyway Automated Schema Migration	950
300.2. Flyway Schema Source	950
300.3. Flyway Automatic Schema Population	951
300.4. Database Server Profiles	951
300.5. Dirty Database Detection	951
300.6. Flyway Migration	952
301. SQL CRUD Commands	953
301.1. H2 Console Access	953
301.2. Postgres CLI Access	953
301.3. Next Value for Sequence	954
301.4. SQL ROW INSERT	954
301.5. SQL SELECT	954
301.6. SQL ROW UPDATE	955
301.7. SQL ROW DELETE	955
301.8. RDBMS Transaction	956
302. JDBC	958
302.1. JDBC DataSource	958
302.2. Obtain Connection and Statement	958
302.3. JDBC Create Example	959
302.4. Set ID Example	960
302.5. JDBC Select Example	960
302.6. nextId	961
302.7. Dialect	962
303. Summary	963
Java Persistence API (JPA)	964
304. Introduction	965
304.1. Goals	965
304.2. Objectives	965
305. Java Persistence API	966

305.1. JPA Standard and Providers	966
305.2. Javax / Jakarta Transition	966
305.3. JPA Dependencies	967
305.4. Enabling JPA AutoConfiguration	968
305.5. Configuring JPA DataSource	969
305.6. Automatic Schema Generation	969
305.7. Schema Generation to File	969
305.8. Generated Schema Output	970
305.9. Other Useful Properties	971
305.10. Configuring JPA Entity Scan	972
305.11. JPA Persistence Unit	973
305.12. JPA Persistence Context	973
306. JPA Entity	975
306.1. JPA @Entity Defaults	975
306.2. JPA Overrides	976
307. Basic JPA CRUD Commands	978
307.1. EntityManager persist()	978
307.2. EntityManager find() By Identity	979
307.3. EntityManager query	979
307.4. EntityManager flush()	980
307.5. EntityManager remove()	981
307.6. EntityManager clear() and detach()	981
308. Transactions	983
308.1. Transactions Required for Explicit Changes/Actions	983
308.2. Activating Transactions	983
308.3. Conceptual Transaction Handling	984
308.4. Activating Transactions in @Components	985
308.5. Calling @Transactional @Component Methods	985
308.6. @Transactional @Component Methods SQL	986
308.7. Unmanaged @Entity	986
308.8. Shared Transaction	987
308.9. @Transactional Attributes	987
309. Summary	989
Spring Data JPA Repository	990
310. Introduction	991
310.1. Goals	991
310.2. Objectives	991
311. Spring Data JPA Repository	992
312. Spring Data Repository Interfaces	993
313. SongsRepository	994
313.1. Song @Entity	994

313.2. SongsRepository	994
314. Configuration	995
314.1. Injection	995
315. CrudRepository	996
315.1. CrudRepository save() New	996
315.2. CrudRepository save() Update Existing	997
315.3. CrudRepository save()/Update Resulting SQL	997
315.4. New Entity?	998
315.5. CrudRepository existsById()	998
315.6. CrudRepository findById()	999
315.7. CrudRepository delete()	1000
315.8. CrudRepository deleteById()	1001
315.9. Other CrudRepository Methods	1001
316. PagingAndSortingRepository	1003
316.1. Sorting	1003
316.2. Paging	1004
316.3. Page Result	1005
316.4. Slice Properties	1005
316.5. Page Properties	1006
316.6. Stateful Pageable Creation	1006
316.7. Page Iteration	1007
317. Query By Example	1008
317.1. Example Object	1008
317.2. findAll By Example	1009
317.3. Primitive Types are Non-Null	1009
317.4. matchingAny ExampleMatcher	1010
317.5. Ignoring Properties	1011
317.6. Contains ExampleMatcher	1011
318. Derived Queries	1013
318.1. Single Field Exact Match Example	1013
318.2. Query Keywords	1014
318.3. Other Keywords	1014
318.4. Multiple Fields	1015
318.5. Collection Response Query Example	1015
318.6. Slice Response Query Example	1016
318.7. Page Response Query Example	1017
319. JPA-QL Named Queries	1019
319.1. Mapping @NamedQueries to Repository Methods	1019
320. @Query Annotation Queries	1021
320.1. @Query Annotation Native Queries	1021
320.2. @Query Sort and Paging	1022

321. JpaRepository Methods	1023
321.1. JpaRepository Type Extensions	1023
321.2. JpaRepository flush()	1024
321.3. JpaRepository deleteInBatch	1025
321.4. JPA References	1025
322. Custom Queries	1028
322.1. Custom Query Interface	1028
322.2. Repository Extends Custom Query Interface	1028
322.3. Custom Query Method Implementation	1028
322.4. Repository Implementation Postfix	1029
322.5. Helper Methods	1029
322.6. Naive Injections	1030
322.7. Required Injections	1030
322.8. Calling Custom Query	1031
323. Summary	1032
323.1. Comparing Query Types	1032
JPA Repository End-to-End Application	1033
324. Introduction	1034
324.1. Goals	1034
324.2. Objectives	1034
325. BO/DTO Component Architecture	1035
325.1. Business Object(s)/@Entities	1035
325.2. Data Transfer Object(s) (DTOs)	1036
325.3. BO/DTO Mapping	1037
326. Service Architecture	1040
326.1. Injected Service Boundaries	1040
326.2. Compound Services	1042
327. BO/DTO Interface Options	1044
327.1. API Maps DTO/BO	1044
327.2. @Service Maps DTO/BO	1044
327.3. Layered Service Mapping Approach	1045
328. Implementation Details	1046
328.1. Song BO	1046
328.2. SongDTO	1046
328.3. Song JSON Rendering	1048
328.4. Song XML Rendering	1048
328.5. Pageable/PageableDTO	1048
328.6. Page/PageDTO	1051
329. SongMapper	1054
329.1. Example Map: SongDTO to Song BO	1054
329.2. Example Map: Song BO to SongDTO	1054

330. Service Tier	1055
330.1. SongsService Interface	1055
330.2. SongsServiceImpl Class	1055
330.3. createSong()	1056
330.4. findSongsMatchingAll()	1056
331. RestController API	1058
331.1. createSong()	1058
331.2. findSongsByExample()	1059
331.3. WebClient Example	1059
332. Summary	1061
MongoDB with Mongo Shell	1062
333. Introduction	1063
333.1. Goals	1063
333.2. Objectives	1063
334. Mongo Concepts	1064
334.1. Mongo Terms	1064
334.2. Mongo Documents	1064
335. MongoDB Server	1066
335.1. Starting Docker Compose MongoDB	1066
335.2. Connecting using Host's Mongo Shell	1066
335.3. Connecting using Guest's Mongo Shell	1066
335.4. Switch to test Database	1067
335.5. Database Command Help	1067
336. Basic CRUD Commands	1069
336.1. Insert Document	1069
336.2. Primary Keys	1069
336.3. Document Index	1069
336.4. Create Index	1070
336.5. Find All Documents	1070
336.6. Return Only Specific Fields	1071
336.7. Get Document by Id	1071
336.8. Replace Document	1071
336.9. Save/Upsert a Document	1072
336.10. Update Field	1072
336.11. Delete a Document	1073
337. Paging Commands	1074
337.1. Sample Documents	1074
337.2. limit()	1074
337.3. sort()/skip()/limit()	1075
338. Aggregation Pipelines	1077
338.1. Common Commands	1077

338.2. Unique Commands	1077
338.3. Simple Match Example	1077
338.4. Count Matches	1078
339. Helpful Commands	1080
339.1. Default Database	1080
339.2. Command-Line Script	1080
340. Summary	1082
MongoTemplate	1083
341. Introduction	1084
341.1. Goals	1084
341.2. Objectives	1084
342. MongoDB Project	1086
342.1. MongoDB Project Dependencies	1086
342.2. MongoDB Project Integration Testing Options	1086
342.3. Flapdoodle Test Dependencies	1087
342.4. Flapdoodle Properties	1088
342.5. MongoDB Access Objects	1088
342.6. MongoDB Connection Properties	1089
342.7. Injecting MongoTemplate	1090
342.8. Disabling Embedded MongoDB	1090
342.9. @ActiveProfiles	1091
342.10. TestProfileResolver	1091
342.11. Using TestProfileResolver	1092
342.12. Inject MongoTemplate	1093
342.13. Testcontainers	1093
343. Example POJO	1094
343.1. Property Mapping	1094
343.2. Field Mapping	1095
343.3. Instantiation	1096
343.4. Property Population	1096
344. Command Types	1097
345. Whole Document Operations	1098
345.1. insert()	1098
345.2. save()/Upsert	1099
345.3. remove()	1100
346. Operations By ID	1101
346.1. findById()	1101
347. Operations By Query Filter	1102
347.1. exists() By Criteria	1102
347.2. delete()	1103
348. Field Modification Operations	1104

348.1. update() Field(s)	1104
348.2. upsert() Fields	1104
349. Paging	1106
349.1. skip()/limit()	1106
349.2. Sort	1106
349.3. Pageable	1107
350. Aggregation	1108
351. ACID Transactions	1109
351.1. Atomicity	1109
351.2. Consistency	1109
351.3. Isolation	1109
351.4. Durability	1110
352. Summary	1111
Spring Data MongoDB Repository	1112
353. Introduction	1113
353.1. Goals	1113
353.2. Objectives	1113
354. Spring Data MongoDB Repository	1114
355. Spring Data MongoDB Repository Interfaces	1115
356. BooksRepository	1116
356.1. Book @Document	1116
356.2. BooksRepository	1116
357. Configuration	1117
357.1. Injection	1117
358. CrudRepository	1118
358.1. CrudRepository save() New	1118
358.2. CrudRepository save() Update Existing	1119
358.3. CrudRepository save()/Update Resulting MongoDB Command	1119
358.4. CrudRepository existsById()	1119
358.5. CrudRepository findById()	1120
358.6. CrudRepository delete()	1121
358.7. CrudRepository deleteById()	1122
358.8. Other CrudRepository Methods	1122
359. PagingAndSortingRepository	1123
359.1. Sorting	1123
359.2. Paging	1124
359.3. Page Result	1124
359.4. Slice Properties	1125
359.5. Page Properties	1125
359.6. Stateful Pageable Creation	1126
359.7. Page Iteration	1126

360. Query By Example	1127
360.1. Example Object	1127
360.2. findAll By Example	1128
360.3. Ignoring Properties	1128
360.4. Contains ExampleMatcher	1129
361. Derived Queries	1130
361.1. Single Field Exact Match Example	1130
361.2. Query Keywords	1131
361.3. Other Keywords	1131
361.4. Multiple Fields	1131
361.5. Collection Response Query Example	1132
361.6. Slice Response Query Example	1132
361.7. Page Response Query Example	1133
362. @Query Annotation Queries	1135
362.1. @Query Annotation Attributes	1135
362.2. @Query Sort and Paging	1136
363. MongoRepository Methods	1137
364. Custom Queries	1138
364.1. Custom Query Interface	1138
364.2. Repository Extends Custom Query Interface	1138
364.3. Custom Query Method Implementation	1138
364.4. Repository Implementation Postfix	1139
364.5. Helper Methods	1139
364.6. Naive Injections	1140
364.7. Required Injections	1140
364.8. Calling Custom Query	1140
364.9. Implementing Aggregation	1141
365. Summary	1142
Mongo Repository End-to-End Application	1143
366. Introduction	1144
366.1. Goals	1144
366.2. Objectives	1144
367. BO/DTO Component Architecture	1145
367.1. Business Object(s)/@Documents	1145
367.2. Data Transfer Object(s) (DTOs)	1146
367.3. BookDTO Class	1147
367.4. BO/DTO Mapping	1147
368. Service Architecture	1150
368.1. Injected Service Boundaries	1150
368.2. Compound Services	1152
369. BO/DTO Interface Options	1154

369.1. API Maps DTO/BO	1154
369.2. @Service Maps DTO/BO	1154
369.3. Layered Service Mapping Approach	1155
370. Implementation Details	1156
370.1. Book BO	1156
370.2. BookDTO	1156
370.3. Book JSON Rendering	1158
370.4. Book XML Rendering	1158
370.5. Pageable/PageableDTO	1158
370.6. Page/PageDTO	1161
371. BookMapper	1164
371.1. Example Map: BookDTO to Book BO	1164
371.2. Example Map: Book BO to BookDTO	1164
372. Service Tier	1165
372.1. BooksService Interface	1165
372.2. BooksServiceImpl Class	1165
372.3. createBook()	1166
372.4. findBooksMatchingAll()	1166
373. RestController API	1168
373.1. createBook()	1168
373.2. findBooksByExample()	1169
373.3. WebClient Example	1169
374. Summary	1171
Heroku Database Deployments	1172
375. Introduction	1173
375.1. Goals	1173
375.2. Objectives	1173
376. Production Properties	1174
376.1. Postgres Production Properties	1174
376.2. Mongo Production Properties	1174
377. Parsing Runtime Properties	1175
377.1. Environment Variable Script	1175
377.2. Script Output	1176
377.3. Heroku DataSource Property	1176
377.4. Testing DATABASE_URL	1177
377.5. MongoDB Properties	1177
377.6. PORT Property	1178
378. Docker Image	1179
378.1. Dockerfile	1179
378.2. Spotify Docker Build Maven Plugin	1179
379. Heroku Deployment	1181

379.1. Provision MongoDB	1181
379.2. Provision Application	1181
379.3. Provision Postgres	1181
379.4. Deploy Application	1182
379.5. Release the Application	1182
380. Summary	1184
AutoRentals Assignment 5: DB	1185
381. Assignment 5a: Spring Data JPA	1186
381.1. Database Schema	1186
381.2. Entity/BO Class	1192
381.3. JPA Repository	1197
382. Assignment 5b: Spring Data Mongo	1201
382.1. Mongo Client Connection	1201
382.2. Mongo Document	1204
382.3. Mongo Repository	1207
383. Assignment 5c: Spring Data Application	1210
383.1. API/Service/DB End-to-End	1210
Bean Validation	1216
384. Introduction	1217
384.1. Goals	1217
384.2. Objectives	1217
385. Background	1219
386. Dependencies	1220
387. Declarative Constraints	1221
387.1. Data Constraints	1221
387.2. Common Built-in Constraints	1221
387.3. Method Constraints	1222
388. Programmatic Validation	1223
388.1. Manual Validator Instantiation	1223
388.2. Inject Validator Instance	1223
388.3. Customizing Injected Instance	1224
388.4. Review: Class with Constraint	1224
388.5. Validate Object	1224
388.6. Validate Method Calls	1225
388.7. Identify Method Using Java Reflection	1226
388.8. Programmatically Check for Parameter Violations	1226
388.9. Validate Method Results	1227
389. Method Parameter Naming	1228
389.1. Option 1: Add -parameters to Java Compiler Command	1228
389.2. Option 2: Add Custom ParameterNameProvider	1229
390. Graphs	1232

390.1. Graph Non-Traversal	1232
390.2. Graph Traversal	1233
391. Groups	1234
391.1. Custom Validation Groups	1234
391.2. Applying Groups	1234
391.3. Skipping Groups	1235
391.4. Applying Groups	1235
392. Multiple Groups	1237
392.1. Example Class with Different Groups	1237
392.2. Validate All Supplied Groups	1237
392.3. Short-Circuit Validation	1238
392.4. Override Default Group	1239
393. Spring Integration	1240
393.1. Validated Component	1240
393.2. ConstraintViolationException	1241
393.3. Successful Validation	1241
393.4. Liskov Substitution Principle	1241
393.5. Disabling Parameter Constraint Override	1242
393.6. Spring Validated Group(s)	1243
393.7. Spring Validated Group(s) Example	1243
394. Custom Validation	1245
394.1. Constraint Interface Definition	1245
394.2. @Documented Annotation	1246
394.3. @Target Annotation	1246
394.4. @Retention	1247
394.5. @Repeatable	1248
394.6. @Constraint	1249
394.7. @MinAge-specific Properties	1250
394.8. Constraint Implementation Class	1250
394.9. Constraint Implementation Type Examples	1251
394.10. Constraint Initialization	1251
394.11. Constraint Validation	1252
394.12. Custom Violation Messages	1252
395. Cross-Parameter Validation	1254
395.1. Cross-Parameter Annotation	1254
395.2. @SupportedValidationTarget	1254
395.3. Method Call Correctness Validation	1255
395.4. Constraint Validation	1255
396. Web API Integration	1257
396.1. Vanilla Spring/AOP Validation	1257
396.2. ConstraintViolationException Not Handled	1257

396.3. ConstraintViolationException Exception Advice	1258
396.4. ConstraintViolationException Mapping Result	1259
396.5. Controller Constraint Validation	1259
396.6. MethodArgumentNotValidException	1260
396.7. MethodArgumentNotValidException Custom Mapping	1260
396.8. @PathVariable Validation	1261
396.9. @PathVariable Validation Result	1262
396.10. @RequestParam Validation	1262
396.11. @RequestParam Validation Violation Response	1263
396.12. Non-Client Errors	1263
396.13. Service Method Error	1264
396.14. Violation Incorrectly Reported as Client Error	1264
396.15. Checking Violation Source	1264
396.16. Internal Server Error Correctly Reported	1265
396.17. Service-detected Client Errors	1266
396.18. Payload	1266
396.19. Exception Handler Checking Payloads	1267
396.20. Internal Violation Exception Handler Results	1268
397. JPA Integration	1269
398. Mongo Integration	1270
398.1. Validating Saves	1270
398.2. ValidatingMongoEventListener	1271
398.3. Other AbstractMongoEventListener Events	1271
398.4. MongoMappingEvent	1272
399. Patterns / Anti-Patterns	1273
399.1. Data Tier Validation	1273
399.2. Use case-specific Validation	1273
399.3. Anti: Validation Everywhere	1274
400. Summary	1276
Porting to Spring Boot 3 / Spring 6	1277
401. Introduction	1278
401.1. Goals	1278
401.2. Objectives	1278
402. Background	1279
403. Preparation	1280
404. Dependency Changes	1281
404.1. Spring Boot Version	1281
404.2. JAXB DependencyManagement	1281
404.3. Groovy	1283
404.4. Spock	1284
404.5. Flapdoodle	1285

404.6. HttpClient / SSL	1286
404.7. Javax /Jakarta Artifact Dependencies	1287
404.8. jakarta.inject	1287
404.9. ActiveMQ / Artemis Dependency Changes	1288
405. Package Changes	1289
406. AssertJ Template Changes	1290
407. Spring Boot Configuration Property	1292
408. HttpStatus	1293
408.1. Spring Boot 2.x	1293
408.2. Spring Boot 3.x	1294
409. HttpMethod	1296
409.1. Spring Boot 2.x	1296
409.2. Spring Boot 3.x	1296
410. Spring Factories Changes	1297
411. Spring WebSecurityConfigurerAdapter	1298
411.1. SecurityFilterChain securityMatcher	1298
411.2. antMatchers/requestMatchers	1299
411.3. ignoringAntMatchers/ignoringRequestMatchers	1300
411.4. authorizeRequests/authorizeHttpRequests	1301
412. Role Hierarchy	1302
412.1. Spring Boot 2.x Role Inheritance	1302
412.2. Spring Boot 3.x Role Inheritance	1302
413. Annotated Method Security	1304
414. @Secured	1305
415. JSR250 RolesAllowed	1306
415.1. Spring Boot 2.x	1306
415.2. Spring Boot 3.x	1306
416. HTTP Client	1307
416.1. Spring Boot 2 HTTP Client	1307
416.2. Spring Boot 3.x HttpClient5	1308
417. Subject Alternative Name (SAN)	1311
418. Swagger Changes	1312
418.1. Spring Doc	1312
418.2. Spring Boot 3.x	1312
419. JPA Dependencies	1313
419.1. Spring Boot 3.x/Hibernate 6.x	1313
420. JPA Default Sequence	1314
420.1. Spring Boot 2.x/Hibernate 5.x	1314
420.2. Spring Boot 3.x/Hibernate 6.x	1314
421. JPA Property Changes	1316
421.1. Spring Boot 2.x/Hibernate 5.x	1316

421.2. Spring Boot 3.x/Hibernate 6.x.....	1316
422. Embedded Mongo.....	1317
422.1. Embedded Mongo AutoConfiguration.....	1317
422.2. Embedded Mongo Properties.....	1317
423. ActiveMQ/Artemis.....	1318
423.1. Spring Boot 2.x.....	1318
423.2. Spring Boot 3.x.....	1318
424. Summary.....	1320
425. Porting to Spring Boot 3.3.2	1321
425.1. JarLauncher	1321
425.2. Role Inheritance	1321
425.3. Flyway 10: Unsupported Database: PostgreSQL	1323
425.4. MongoHealthIndicator.....	1323
425.5. Spring MVC @Validated	1324
JWT/JWS Token Authn/Authz	1326
426. Introduction	1327
426.1. Goals	1327
426.2. Objectives	1327
427. Identity and Authorities	1328
427.1. BASIC Authentication/Authorization	1328
428. Tokens	1330
428.1. Token Authentication/Login	1330
428.2. Token Authorization/Operation	1331
428.3. Authentication Separate from Authorization	1332
428.4. JWT Terms	1332
429. JWT Authentication	1334
429.1. Example JWT Authentication/Login Flow	1334
429.2. Example JWT Authorization/Operation Call Flow	1334
430. Maven Dependencies	1336
431. JwtConfig	1337
431.1. JwtConfig application.properties	1337
432. JwtUtil	1339
432.1. Dependencies on JwtUtil	1339
432.2. JwtUtil: generateToken()	1340
432.3. JwtUtil: generateToken() Helper Methods	1340
432.4. Example Encoded JWS	1341
432.5. Example Decoded JWS Header and Body	1341
432.6. JwtUtil: parseToken()	1342
432.7. JwtUtil: parseToken() Helper Methods	1343
433. JwtAuthenticationFilter	1344
433.1. JwtAuthenticationFilter Relationships	1344

433.2. JwtAuthenticationFilter: Constructor	1345
433.3. JwtAuthenticationFilter: attemptAuthentication()	1345
433.4. JwtAuthenticationFilter: attemptAuthentication() DTO	1346
433.5. JwtAuthenticationFilter: attemptAuthentication() Helper Method	1347
433.6. JwtAuthenticationFilter: successfulAuthentication()	1347
434. JwtAuthorizationFilter	1349
434.1. JwtAuthorizationFilter Relationships	1349
434.2. JwtAuthorizationFilter: Constructor	1350
434.3. JwtAuthorizationFilter: doFilterInternal()	1350
434.4. JwtAuthenticationToken	1351
434.5. JwtEntryPoint	1353
435. API Security Configuration	1354
435.1. API Authentication Manager Builder	1354
435.2. API HttpSecurity Key JWS Parts	1355
435.3. API HttpSecurity Full Details	1355
436. Example JWT/JWS Application	1357
436.1. Roles and Role Inheritance	1357
436.2. CartsService	1357
436.3. Login	1358
436.4. createCart()	1359
436.5. addItem()	1360
436.6. getCart()	1361
436.7. removeCart()	1362
437. Summary	1364
Unit Integration Testing	1365
438. Introduction	1366
438.1. Goals	1366
438.2. Objectives	1367
439. Votes and Elections Service	1368
439.1. Main Application Flows	1368
439.2. Service Event Integration	1368
440. Physical Architecture	1370
440.1. Unit Integration Test Physical Architecture	1370
441. Mongo Integration	1372
441.1. MongoDB Maven Dependencies	1372
441.2. Test MongoDB Maven Dependency	1372
441.3. MongoDB Properties	1373
441.4. MongoDB Repository	1373
441.5. VoteDTO MongoDB Document Class	1374
441.6. Sample MongoDB/VoterRepository Calls	1374
442. ActiveMQ Integration	1376

442.1. ActiveMQ Maven Dependencies	1376
442.2. ActiveMQ Unit Integration Test Properties	1377
442.3. Service Joinpoint Advice	1377
442.4. JMS Publish	1378
442.5. ObjectMapper	1379
442.6. JMS Receive	1379
442.7. EventListener	1380
443. JPA Integration	1381
443.1. JPA Core Maven Dependencies	1381
443.2. JPA Test Dependencies	1381
443.3. JPA Properties	1381
443.4. Database Schema Migration	1382
443.5. Flyway RDBMS Schema Migration	1382
443.6. Flyway RDBMS Schema Migration Files	1383
443.7. Flyway RDBMS Schema Migration Output	1384
443.8. JPA Repository	1384
443.9. Example VoteBO Entity Class	1384
443.10. Sample JPA/ElectionRepository Calls	1385
444. Unit Integration Test	1387
444.1. ClientTestConfiguration	1387
444.2. Example Test	1388
445. Summary	1389

Chapter 1. Abstract

This book contains course notes covering Enterprise Computing with Java. This comprehensive course explores core application aspects for developing, configuring, securing, deploying, and testing a Java-based service using a layered set of modern frameworks and libraries that can be used to develop full services and microservices to be deployed within a container. The emphasis of this course is on the center of the application (e.g., Spring, Spring Boot, Spring Data, and Spring Security) and will lay the foundation for other aspects (e.g., API, SQL and NoSQL data tiers, distributed services) covered in related courses.

Students will learn thru lecture, examples, and hands-on experience in building multi-tier enterprise services using a configurable set of server-side technologies.

Students will learn to:

- Implement flexibly configured components and integrate them into different applications using inversion of control, injection, and numerous configuration and auto-configuration techniques
- Implement unit and integration tests to demonstrate and verify the capabilities of their applications using JUnit and Spock
- Implement basic API access to service logic using using modern RESTful approaches that include JSON and XML
- Implement basic data access tiers to relational and NoSQL databases using the Spring Data framework
- Implement security mechanisms to control access to deployed applications using the Spring Security framework

Using modern development tools students will design and implement several significant programming projects using the above-mentioned technologies and deploy them to an environment that they will manage.

The course is continually updated and currently based on Java 11, Spring 5.x, and Spring Boot 2.x.

Enterprise Computing with Java (605.784.8VL) Course Syllabus

copyright © 2024 jim stafford (jim.stafford@jhu.edu)

Chapter 2. Course Description

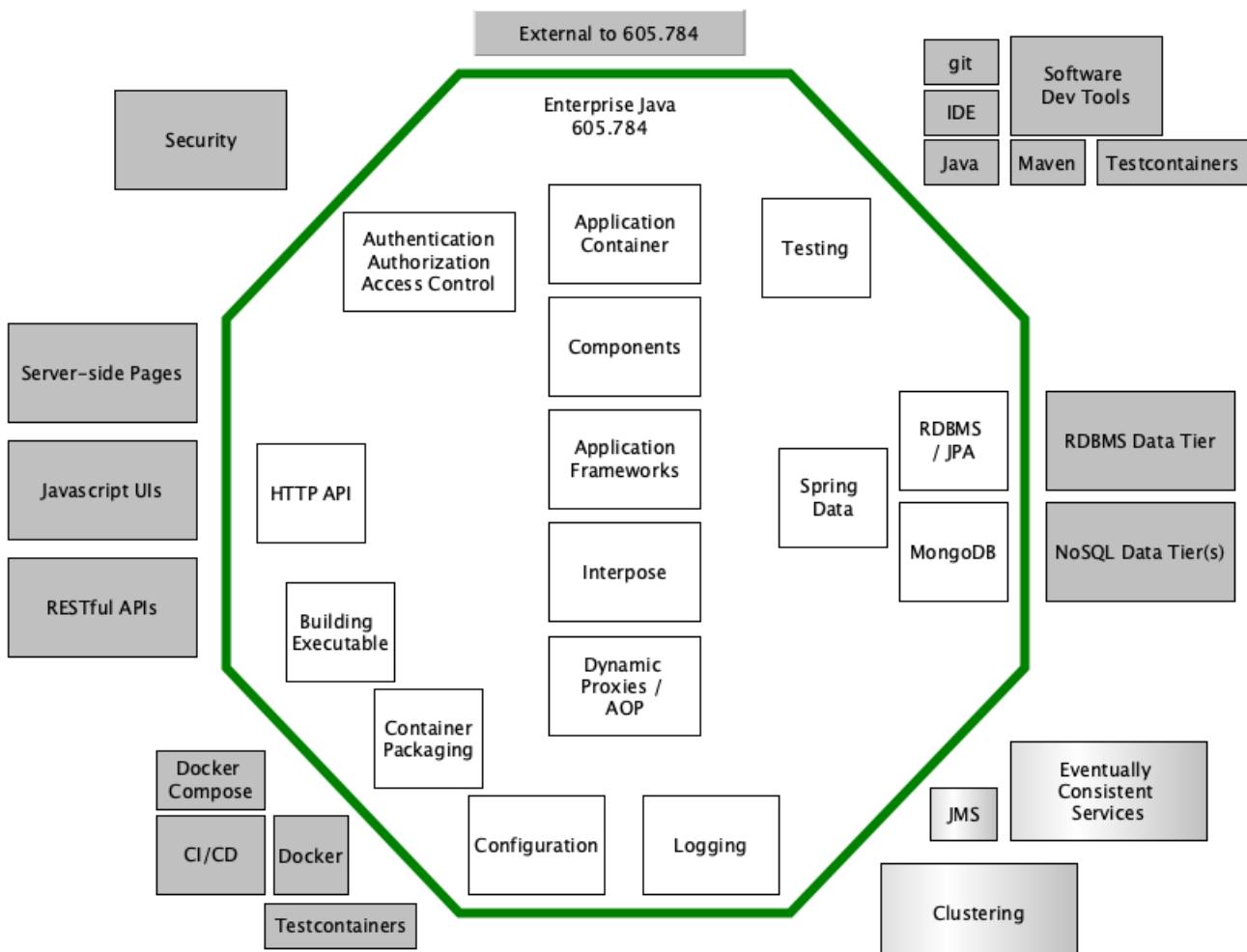
2.1. Meeting Times/Location

- Wednesdays, 4:30-7:10pm EST
 - onsite: APL K219
 - via Zoom [Meeting ID: 969 9489 2663](#)
- [Canvas](#)
- [Git Repository](#)

2.2. Course Goal

The goal of this course is to master the design and development challenges of a single application instance to be deployed in an enterprise-ready Java application framework. This course provides the bedrock for materializing broader architectural solutions within the body of a single instance.

Course Topic External Relationships



2.3. Description

This comprehensive course explores core application aspects for developing, configuring, securing, deploying, and testing a Java-based service using a layered set of modern frameworks and libraries that can be used to develop full services and microservices to be deployed within a container. The emphasis of this course is on the center of the application (e.g., Spring, Spring Boot, Spring Data, and Spring Security) and will lay the foundation for other aspects (e.g., API, SQL and NoSQL data tiers, distributed services) covered in related courses.

Students will learn thru lecture, examples, and hands-on experience in building multi-tier enterprise services using a configurable set of server-side technologies.

Students will learn to:

- Implement flexibly configured components and integrate them into different applications using inversion of control, injection, and numerous configuration and auto-configuration techniques
- Implement unit and integration tests to demonstrate and verify the capabilities of their applications using JUnit
- Implement API access to service logic using using modern RESTful approaches that include JSON and XML
- Implement data access tiers to relational and NoSQL (MongoDB) databases using the Spring Data framework
- Implement security mechanisms to control access to deployed applications using the Spring Security framework
- Package, run, and test services within a Docker container

Using modern development tools students will design and implement several significant programming projects using the above-mentioned technologies in an environment that they will manage.

The course is continually updated and currently based on Java 17, Maven 3, Spring 6.x, and Spring Boot 3.x.

2.4. Student Background

- Prerequisite: 605.481 Distributed Development on the World Wide Web or equivalent
- Strong Java programming skills are assumed
- Familiarity with Maven and IDEs is helpful
- Familiarity with Docker (as a user) can be helpful in setting up a local development environment quickly

2.5. Student Commitment

- Students should be prepared to spend between 6-10 hours a week outside of class. Time spent can be made efficient by proactively keeping up with class topics and actively collaborating

with the instructor and other students in the course.

2.6. Course Text(s)

The course uses no mandatory text. The course comes with many examples, course notes for each topic, and references to other free Internet resources.

2.7. Required Software

Students are required to establish a local development environment.

1. Software you will need to load onto your local development environment:
 - a. Git Client
 - b. Java JDK 17
 - c. Maven 3 ($\geq 3.6.3$)
 - d. IDE (IntelliJ IDEA Community Edition or Pro or Eclipse/STS)
 - The instructor will be using IntelliJ IDEA CE in class, but Eclipse/STS is also a good IDE option. **It is best to use what you are already comfortable using.**
 - e. JHU VPN (Open Pulse Secure) — workarounds are available
2. Software you will ideally load onto your local development environment:
 - a. Docker
 - Docker can be used to automate software installation and setup and implement deployment and integration testing techniques. Several pre-defined images, ready to launch, will be made available in class.
 - b. curl or something similar
 - c. Postman API Client or something similar
3. Software you will need to install **if** you do not have Docker
 - a. MongoDB
4. High visibility software you will use that will get downloaded and automatically used through Maven.
 - a. application framework (Spring Boot, Spring).
 - b. SLF/Logback
 - c. a relational database (H2 Database Engine) and JPA persistence provider (Hibernate)
 - d. JUnit
 - e. Testcontainers

2.8. Course Structure

Lectures are conducted live each week and reflect recent/ongoing student activity. Students may optionally attend the lecture live online and/or watch the recording based on their personal

schedule. There is no required attendance for the live lecture.

The course materials consist of a large set of examples that you will download, build, and work with locally. The course also provides a set of detailed course notes for each lecture and an associated assignment active at all times during the semester. Topics and assignments have been grouped into application development, service/API tier, containers, and data tier. Each group consists of multiple topics that span one or more weeks.

The examples are available in a Gitlab public repository. The course notes are available in HTML and PDF format for download. All content or links to content is published on the course public website (<https://jcs.ep.jhu.edu/ejava-springboot>). To help you locate and focus on current content and not be overwhelmed with the entire semester, examples and links to content are activated as the semester progresses. A list of "What is new" and "Student TODOs" is published weekly before class to help you keep up to date and locate relevant material. The complete set of content from the previous semester is always available from the legacy link (<https://jcs.ep.jhu.edu/legacy-ejava-springboot/>)

2.9. Grading

- 100 >= A >= 90 > B >= 80 > C >= 70 > F

Assessment	% of Semester Grade
Class/Newsgroup Participation	10% (9pm EST, Wed weekly cut-off)
Assignment 0: Application Build	5% (##)
Assignment 1: Application Config	20%
Assignment 2: Web API	15%
Assignment 3: Security	15%
Assignment 4: Integration Testing and Containers	10%
Assignment 5: Database	25%



Do not host your course assignments in a public Internet repository.

Course assignments should not be posted in a public Internet repository. If using an Internet repository, only the instructor should have access.

- Assignments will be done individually and most are graded 100 though 0, based on posted project grading criteria.
 - ## Assignment 0 will be graded on a done (100)/not-done(0) basis and must be turned in on-time in order to qualify for a REDO. The intent of this requirement is to promote early activity with development and early exchange of questions/answers and artifacts between students and instructor.
- Class/newsgroup participation will be based on instructor judgment whether the student has made a contribution to class to either the classroom or newsgroup on a consistent weekly basis. A newsgroup contribution may be a well-formed technical observation/lesson learned, a well

formed question that leads to a well formed follow up from another student, or a well formed answer/follow-up to another student's question. Well formed submissions are those that clearly summarize the topic in the subject, and clearly describe the objective, environment, and conditions in the body. The instructor will be the judge of whether a newsgroup contribution meets the minimum requirements for the week. The intent of this requirement is to promote active and open collaboration between class members.

- Weekly cut-off for newsgroup contributions is each Wed @9pm EST

2.10. Grading Policy

- Late assignments will be deducted 10pts/week late, starting after the due date/time, with a one-time exception. A student may submit a single project up to 4 days late without receiving approval and still receive complete credit. Students taking advantage of the "free first pass" should still submit an e-mail to the instructor and grader(s) notifying them of their intent.
- Class attendance is strongly recommended, but not mandatory. The student is responsible for obtaining any written or oral information covered during their absence. Each session will be recorded. A link to the recording will be posted on Canvas.

2.11. Academic Integrity

Collaboration of ideas and approaches are strongly encouraged. You may use partial solutions provided by others as a part of your project submission. However, the bulk usage of another students implementation or project will result in a 0 for the project. There is a difference between sharing ideas/code snippets and turning in someone else's work as your own. When in doubt, document your sources.

Do not host your course assignments in a **public** Internet repository.

2.12. Instructor Availability

I am available at least 20min before class, breaks, and most times after class for extra discussion. I monitor/respond to e-mails and the newsgroup discussions and hold ad-hoc office hours via Zoom in the evening and early morning hours plus weekends.

I provide detailed answers to assignment and technical questions through the course newsgroup. You can get individual, non-technical questions answered via the instructor email.

2.13. Communication Policy

I provide detailed answers to assignment and technical questions through the course newsgroup. You can get individual, non-technical questions answered via email but please direct all technical and assignment questions to the newsgroup. If you have a question or make a discovery—it is likely pertinent to most of the class and you are the first to identify.

- Newsgroup: [Canvas Course Discussions]
- Instructor Email: jim.stafford@jhu.edu

I typically respond to all e-mails and newsgroup posts in the evening and early morning hours. Rarely will a response take longer than 24 hours. It is very common for me to ask for a copy of your broken project so that I can provide more analysis and precise feedback. This is commonly transmitted either as an e-mail attachment, a link to a branch in a **private** repository, or an early submission in Canvas.

2.14. Office Hours

Students needing further assistance are welcome to schedule a web meeting using [Zoom Conferencing](#). Most conference times will be between 8 and 10pm EST and 6am to 5pm EST weekends.

Chapter 3. Course Assignments

3.1. General Requirements

- Assignments must be submitted to Canvas with source code in a standard archive file. "target" directories with binaries are not needed and add unnecessary size.
- All assignments must be submitted with a README that points out how the project meets the assignment requirements.
- All assignments must be written to build and run in the grader's environment in a portable manner using [Maven 3](#). This will be clearly spelled out during the course and you may submit partial assignments early to get build portability feedback (not early content grading feedback).
- Test Cases must be written using JUnit 5 and run within the Maven surefire and failsafe environments.
- The course repository will have an assignment-support and assignment-starter set of modules.
 - The assignment-support modules are to be referenced as a dependency and not cloned into student submissions.
 - The assignment-starter modules are skeletal examples of work to be performed in the submitted assignment.

3.2. Submission Guidelines

You should test your application prior to submission by

1. **Verify that your project does not require a pre-populated database.** All setup must come through automated test setup.

This will make sure you are not depending on any residue schema or data in your database.

2. **Run maven clean and archive your project from the root** without pre-build target directory files.

This will help assure you are only submitting source files and are including all necessary source files within the scope of the assignment.

3. **Move your localRepository** (or set your settings.xml#localRepository value to a new location — do not delete your primary localRepository)

This will hide any old module SNAPSHOTs that are no longer built by the source (e.g., GAV was changed in source but not sibling dependency).

4. **Explode the archive in a new location and run mvn clean install from the root** of your project.

This will make sure you do not have dependencies on older versions of your modules or manually installed artifacts. This, of course, will download all project dependencies and help verify that the project will build in other environments. This will also simulate what the grader

will see when they initially work with your project.

5. Make sure the **README documents all information required to demonstrate or navigate your application** and point out issues that would be important for the evaluator to know (e.g., "the instructor said...")

Chapter 4. Syllabus

Table 1. Core Development

#	Date	Lectures	Assignments/Notes
1	Aug28	Course Introduction <ul style="list-style-type: none"> Intro to Enterprise Java Frameworks notes 	<ul style="list-style-type: none"> Devenv Setup spec
		Spring/Spring Boot Introduction <ul style="list-style-type: none"> Pure Java Main Application notes Spring Boot Application notes 	<ul style="list-style-type: none"> Assignment 0 App Build spec Due: Tue Sep03, 6am
2	Sep04	Spring Boot Configuration <ul style="list-style-type: none"> Bean Factory and Dependency Injection notes Value Injection notes Property Sources notes Configuration Properties notes 	<ul style="list-style-type: none"> Assignment 1a App Config spec Due: Wed Sep25, 6am
3	Sep11	<ul style="list-style-type: none"> Auto-Configuration notes 	
		Logging notes	<ul style="list-style-type: none"> Assignment 1b Logging spec Due: Wed Sep25, 6am
4	Sep18	Testing notes	<ul style="list-style-type: none"> Assignment 1c Testing spec Due: Wed Sep25, 6am

Table 2. Service and API Tiers

#	Date	Lectures	Assignments/Notes
4	Sep18 (Cont)	API <ul style="list-style-type: none"> HTTP-based/REST-like API notes 	<ul style="list-style-type: none"> Assignment 2 API spec Due: Sun Oct13, 6am
5	Sep25	API (Cont) <ul style="list-style-type: none"> Spring MVC notes Controller/Service Interface notes 	
6	Oct02	<ul style="list-style-type: none"> Data/Content Marshalling notes API Documentation notes 	

#	Date	Lectures	Assignments/Notes
7	Oct09	Spring Security <ul style="list-style-type: none"> • Spring Security Introduction notes • Authentication notes 	<ul style="list-style-type: none"> • Assignment 3a Security spec Due: Sun Nov03, 6am
8	Oct16	<ul style="list-style-type: none"> • User Details notes • Authorization/Access Control notes • Enabling HTTPS/TLS notes 	<ul style="list-style-type: none"> • Assignment 3b Authorization spec • Assignment 3c HTTPS spec Due: Sun Nov03, 6am
9	Oct23	AOP and Method Proxies notes	<ul style="list-style-type: none"> • Assignment 3d AOP spec Due: Sun Nov03, 6am

Table 3. Data Tier

#	Date	Lectures	Assignments/Notes
10	Oct30	Integration Testing and Containers <ul style="list-style-type: none"> • Maven Integration Test notes • Docker Images notes • Docker IT notes • [Docker Compose notes] • [Docker Compose IT notes] • Testcontainers NTest notes 	<ul style="list-style-type: none"> • Assignment 4 Integration Testing and Containers spec Due: Sun Nov17, 6am
11	Nov06	JPA Mapping <ul style="list-style-type: none"> • RDBMS notes • Java Persistence API (JPA) notes 	<ul style="list-style-type: none"> • Assignment 5a Spring Data JPA spec Due: Sun, Dec08, 6am
12	Nov13	Spring Data JPA Repository notes MongoDB NoSQL Mapping <ul style="list-style-type: none"> • MongoDB and Shell notes • MongoTemplate notes 	<ul style="list-style-type: none"> • Assignment 5b Spring Data Mongo spec Due: Sun Dec08, 6am
13	Nov20	Spring Data MongoDB Repository notes <ul style="list-style-type: none"> • Spring Data End-to-End notes notes	<ul style="list-style-type: none"> • Assignment 5c Spring Data End-to-End spec Due: Sun Dec08, 6am
	Nov27	Thanksgiving	no class
14	Dec04	Validation notes	

Table 4. Other Topics

		<ul style="list-style-type: none">• [Porting to Spring Boot 3/Spring 6] notes• [JSON Web Tokens] notes• [Integration Unit Test] notes• [Testcontainers with JUnit] notes• [Testcontainers with Spock] notes	
--	--	--	--

Development Environment Setup

copyright © 2024 jim stafford (jim.stafford@jhu.edu)

Chapter 5. Introduction

Participation in this course requires a local development environment. Since competence using Java is a prerequisite for taking the course, much of the content here is likely already installed in your environment.

Software versions do not have to be latest-and-greatest. For the most part, the firm requirement is that the JDK must be 17 or at least your source code needs to stick to Java 17 features to be portable to grading environments.

You must manually download and install some of the software locally (e.g., IDE). Some software installations (e.g., MongoDB) have simple Docker options. The remaining set will download automatically and run within Maven. Some software is needed day 1. Others can wait.

Rather than repeat detailed software installation procedures for the various environments, I will list each one, describe its purpose in the class, and direct you to one or more options to obtain. Please make use of the course newsgroup if you run into trouble or have questions.

5.1. Goals

The student will:

- setup required tooling in local development environment and be ready to work with the course examples

5.2. Objectives

At the conclusion of these instructions, the student will have:

1. installed Java JDK 17
2. installed Maven 3
3. installed a Git Client and checked out the course examples repository
4. installed a Java IDE (IntelliJ IDEA Community Edition, Eclipse Enterprise, or Eclipse/STS)
5. installed a Web API Client tool
6. optionally installed Docker
7. conditionally installed Mongo

Chapter 6. Software Setup

6.1. Java JDK (immediately)

You will need a JDK 17 compiler and its accompanying JRE environment immediately in class. Everything we do will revolve around a JVM.

- For Mac and Unix-like platforms, [SDKMan](#) is a good source for many of the modern JDK images. You can also use brew or your package manager (e.g., yum, apt).

```
> brew search jdk | grep 17
openjdk@17

$ sdk list java | egrep 'ms|open|amzn' | grep 17
      | 17.0.12      | amzn      | installed   | 17.0.12-amzn
      | 17.0.11      | amzn      |             | 17.0.11-amzn
      | 17.0.11      | ms        |             | 17.0.11-ms
```

```
# apt-cache search openjdk-17 | egrep 'jdk |jre '
openjdk-17-jdk - OpenJDK Development Kit (JDK)
openjdk-17-jre - OpenJDK Java runtime, using Hotspot JIT
```

- For Windows Users - [Microsoft has JDK images](#) available for direct download. These are the same downloads that SDKMan uses when using the `ms` option.

```
Windows x64 zip microsoft-jdk-17.0.3-windows-x64.zip    sha256 / sig
Windows x64 msi microsoft-jdk-17.0.3-windows-x64.msi    sha256
```

After installing and placing the `bin` directory in your PATH, you should be able to execute the following commands and output a version 17.x of the JRE and compiler.

Example Java Version Check

```
$ java --version
openjdk 17.0.12 2024-07-16 LTS
OpenJDK Runtime Environment Corretto-17.0.12.7.1 (build 17.0.12+7-LTS)
OpenJDK 64-Bit Server VM Corretto-17.0.12.7.1 (build 17.0.12+7-LTS, mixed mode,
sharing)

$ javac --version
javac 17.0.12
```

6.2. Git Client (immediately)

You will need a Git client immediately in class. Note that most IDEs have a built-in/internal Git client capability, so the command line client shown here may not be absolutely necessary. If you chose to use your built-in IDE Git client, just translate any command-line instructions to GUI commands. If you have git already installed, it is highly likely you do not need to upgrade.

Download and install a Git Client.

- All platforms - [Git-SCM](#)



I have git installed through brew on macOS.



If you already have git installed and working — no need for upgrade to latest.

Example git Version Check

```
$ git --version  
git version 2.45.2
```

Checkout the course baseline.

```
$ git clone https://gitlab.com/ejava-javaee/ejava-springboot.git  
...  
  
$ ls | sort  
...  
assignment-starter  
assignment-support  
build  
common  
...  
pom.xml
```

Each week you will want to update your copy of the examples as I updated and release changes.

```
$ git checkout main    # switches to main branch  
$ git pull            # merges in changes from origin
```

Updating Changes to Modified Directory

If you have modified the source tree, you can save your changes to a new branch using the following:



```
$ git status          #show me which files I changed  
$ git diff            #show me what the changes were  
$ git checkout -b new-branch  #create new branch
```

```
$ git commit -am "saving my stuff" #commit my changes to new branch  
$ git checkout main      #switch back to course baseline  
$ git pull                #get latest course examples/corrections
```

Saving Modifications to an Existing Branch

If you have made modifications to the source tree while in the wrong branch, you can save your changes in an existing branch using the following:



```
$ git stash          #save my changes in a temporary area  
$ git checkout existing-branch    #go to existing branch  
$ git commit -am "saving my stuff" #commit my changes to existing  
branch  
$ git checkout main      #switch back to course baseline  
$ git pull                #get latest course examples/corrections
```

6.3. Maven 3 (immediately)

You will need Maven immediately in class. We use Maven to create repeatable and portable builds in class. This software build system is rivaled by Gradle. However, everything presented in this course is based on Maven and there is no feasible way to make that optional.

Download and install Maven 3.

- All platforms - [Apache Maven Project](#)



I have Maven installed manually from the site since `brew` would only install the latest and I had an issue with 3.9.8 that reverting to 3.8.8 fixed. The issue is likely unique to my docs area and should not be an issue with the class examples. Just know that I am currently using 3.8.8 and the examples are built in the CI/CD environment with 3.8.6.

Place the `$MAVEN_HOME/bin` directory in your `$PATH` so that the `mvn` command can be found.

Example Maven Version Check

```
$ mvn --version  
Apache Maven 3.8.8 (4c87b05d9aedce574290d1acc98575ed5eb6cd39)  
Maven home: /usr/local/opt/apache-maven-3.8.8  
Java version: 17.0.12, vendor: Amazon.com Inc., runtime:  
/Users/jim/.sdkman/candidates/java/17.0.12-amzn  
Default locale: en_US, platform encoding: UTF-8  
OS name: "mac os x", version: "12.6.9", arch: "x86_64", family: "mac"
```

Setup any custom settings in `$HOME/.m2/settings.xml`. This is an area where you and I can define environment-specific values referenced by the build.

```

<?xml version="1.0"?>
<settings xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/settings-1.0.0.xsd">

  <!-- keep as default if using class docker-compose script
      <localRepository>somewhere_else</localRepository>
  -->
  <offline>false</offline>

  <mirrors>
    <!-- uncomment when JHU VPN unavailable
    <mirror>
      <id>ejava-nexus</id>
      <mirrorOf>ejava-nexus, ejava-nexus-snapshots, ejava-nexus-
releases</mirrorOf>
      <url>file://${user.home}/.m2/repository/</url>
    </mirror>
    --> ① ②
  </mirrors>

  <profiles>
  </profiles>

  <activeProfiles>
    <!--
    <activeProfile>aProfile</activeProfile>
    -->
  </activeProfiles>

</settings>

```

- ① make sure your ejava-springboot repository:main branch is up to date and installed (i.e., `mvn clean install -f ./build; mvn clean install`) prior to using local mirror
- ② the URL in the mirror must be consistent with the `localRepository` value. The value shown here assumes the default, `$HOME/.m2/repository` value.

Attempt to build the source tree. Report any issues to the course newsgroup.

```

$ pwd
.../ejava-springboot
$ mvn install -f build
...
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
$ mvn clean install

```

```
...
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
```

6.4. Java IDE (immediately)

You will realistically need a Java IDE very early in class. If you are a die-hard vi, emacs, or text editor user—you can do a lot with your current toolset and Maven. However, when it comes to code refactoring, inspecting framework API classes, and debugging, there is no substitute for a good IDE. I have used Eclipse/STS for many years and some students in previous semester have used Eclipse installations from a previous Java-development course or work experience. They are free and work well. I will actively be using IntelliJ IDEA Community Edition. The community edition is free and contains most of the needed support. The professional edition is available for 1 year to anyone supplying a [.edu](#) e-mail.



It is up to you what IDE you use. **Using something familiar is always the best first choice.**

Download and install an IDE for Java development.

- IntelliJ IDEA Community Edition
 - All platforms - [Jetbrains IntelliJ](#)
- Eclipse/STS
 - All platforms - [Spring.io /tools](#)
- Eclipse/Enterprise Java and Web Developers (or whatever...)
 - All platforms - [eclipse.org](#)

Load an attempt to run the examples in

- [app/app-build/java-app-example](#)

6.5. Web API Client tool (not immediately)

Within the first month of the course, it will be helpful for you to have a client that can issue POST, PUT, and DELETE commands in addition to GET commands over HTTP. This will not be necessary until a few weeks into the semester.

Some options include:

- curl - command line tool popular in Unix environments and likely available for Windows. All of my Web API call examples are done using curl.
- [Postman API Client](#) - a UI-based tool for issuing and viewing web requests/responses. I personally do not like how "enterprisey" Postman has become. It used to be a simple browser plugin tool. However, the free version works and seems to only require a sign-up login.

curl example

```
$ curl -v -X GET https://ep.jhu.edu/
<!DOCTYPE html>
<html class="no-js" lang="en">
    <head>
    ...
<title>Johns Hopkins Engineering | Part-Time & Online Graduate Education</title>
    ...
```

6.6. Optionally Install Docker (not immediately)

We will cover and make use of Docker since it is highly likely that anything you develop professionally will be deployed with Docker or will leverage it in some way. Spring/Spring Boot has also embraced Docker and Testcontainers as their preferred integration environment. There will be time-savings setup of backend databases as well as options in assignments where Docker desired. Once the initial investment of installing Docker has been tackled—software deployments, installation, and executions become very portable and easy to achieve.



I am still a bit hesitant in requiring Docker for the class. I will make it optional for the students who cannot install. I will lean into Docker more heavily if I get a sense that all students have access. Let me know where you stand on this optional installation.

Docker can serve three purposes in class:

1. automates example database and JMS resource setup
2. provides a popular example deployment packaging
3. provides an integration test platform option with Maven plugins or with Testcontainers

Without Docker installation, you will

1. need to manually install MongoDB native to your environment
2. be limited to conceptual coverage of deployment and testing options in class

Optionally download and install Docker (called "Docker Desktop" these days).

- All platforms - [Docker.com](https://www.docker.com)
- Also install - [docker-compose](https://github.com/docker/compose)

Example Docker Version Check

```
$ docker -v
Docker version 27.0.3, build 7d4bcd8
$ docker-compose -v
Docker Compose version v2.28.1-desktop.1
```



The Docker Compose capability is now included with Docker via a Docker plugin and executed using `docker compose` versus requiring a separate `docker-compose` wrapper. Functionally they are the same. However, Testcontainers still looks for the legacy `docker-compose` command when testing with Docker Compose—so the wrapper is still necessary.

Legacy docker-compose Wrapper

```
$ docker compose --help
Usage: docker compose [OPTIONS] COMMAND
Define and run multi-container applications with Docker
...
$ docker-compose --help
Usage: docker compose [OPTIONS] COMMAND
Define and run multi-container applications with Docker
...
```

6.6.1. docker-compose Test Drive

With the course baseline checked out, you should be able to perform the following. Your results for the first execution will also include the download of images.

Start up database resources

```
$ docker-compose -p ejava up -d mongodb postgres ①②
Creating ejava_postgres_1 ... done
Creating ejava_mongodb_1 ... done
```

① `-p` option sets the project name to a well-known value (directory name is default)

② `up` starts services and `-d` runs them all in the background

Shutdown database resources

```
$ docker-compose -p ejava stop mongodb postgres ①
Stopping ejava_mongodb_1 ... done
Stopping ejava_postgres_1 ... done
$ docker-compose -p ejava rm -f mongodb postgres ②
Going to remove ejava_mongodb_1, ejava_postgres_1
Removing ejava_mongodb_1 ... done
Removing ejava_postgres_1 ... done
```

① `stop` pauses the running container

② `rm` removes state assigned to the stopped container. `-f` does not request confirmation.

6.7. MongoDB (later)

You will need MongoDB in the later 1/3 of the course. It is somewhat easy to install locally, but a mindless snap—configured exactly the way we need it to be—if we use Docker. Feel free to activate a free [Atlas](#) account if you wish, but what gets turned in for grading should either use a **standard local URL** (using Fongo (via Maven), Docker, or Testcontainers).

If you have not and will not be installing Docker, you will need to install and set up a local instance of Mongo.

- All platforms - [MongoDB](#)

Introduction to Enterprise Java Frameworks

copyright © 2024 jim stafford (jim.stafford@jhu.edu)

Chapter 7. Introduction

7.1. Goals

The student will learn:

- constructs and styles for implementing code reuse
- what is a framework
- what has enabled frameworks
- a historical look at Java frameworks

7.2. Objectives

At the conclusion of this lecture, the student will be able to:

1. identify the key difference between a library and framework
2. identify the purpose for a framework in solving an application solution
3. identify the key concepts that enable a framework
4. identify specific constructs that have enabled the advance of frameworks
5. identify key Java frameworks that have evolved over the years

Chapter 8. Code Reuse

Code reuse is the use of existing software to create new software.^[1]

We leverage code reuse to help solve either repetitive or complex tasks so that we are not repeating ourselves, we reduce errors, and we achieve more complex goals.

8.1. Code Reuse Trade-offs

On the positive side, we do this because we have confidence that we can delegate a portion of our job to code that has been proven to work. We should not need to again test what we are using.

On the negative side, reuse can add dependencies bringing additional size, complexity, and risk to our solution. *If all you need is a spoon — do you need to bring the entire kitchen?*

8.2. Code Reuse Constructs

Code reuse can be performed using several structural techniques

Method Call

We can wrap functional logic within a method within our own code base. We can make calls to this method from the places that require that same task performed.

Classes

We can capture state and functional abstractions in a set of classes. This adds some modularity to related reusable method calls.

Interfaces

Abstract interfaces can be defined as placeholders for things needed but supplied elsewhere. This could be because of different options provided or details being supplied elsewhere.

Modules

Reusable constructs can be packaged into separate physical modules so that they can be flexibly used or not used by our application.

8.3. Code Reuse Styles

There are two basic styles of code reuse and they primarily have to with **control**.

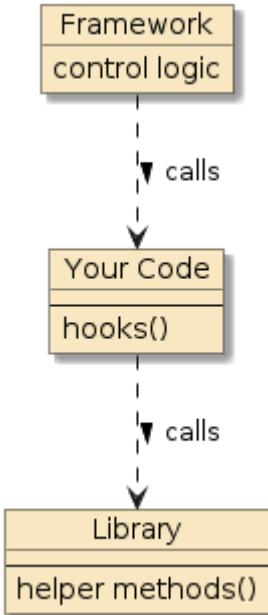


Figure 1. Library/ Framework/Code Relationship ^[2]

Its not always a one-or-the-other style. Libraries can have mini frameworks within them. Even the JSON/XML parser example can be a mini-framework of customizations and extensions.

Libraries

Libraries are modules of reusable code that are invoked on-demand by your code base. Your code is in total control of the library call flow.

- Examples: JSON or XML parser

Frameworks

Frameworks are different from callable libraries—in that they provide some level of control or orchestration. Your code base is under the control of the framework. This is called "**Inversion of Control**".

- Examples: Spring/Spring Boot, JakartaEE (formerly JavaEE)

[1] "Code reuse","Wikipedia"

Chapter 9. Frameworks

9.1. Framework Informal Description

A successful software framework is a body code that has been developed from the skeletons of successful and unsuccessful solutions of the past and present within a common domain of challenge. A framework is a generalization of solutions that provides for key abstractions, opportunity for specialization, and supplies default behavior to make the on-ramp easier and also appropriate for simpler solutions.

- *"We have done this before. This is what we need and this is how we do it."*

A framework is much bigger than a pattern instantiation. A pattern is commonly at the level of specific object interactions. We typically have created or commanded something at the completion of a pattern — but we have a long way to go in order to complete our overall solution goal.

- *Pattern Completion: "that is not enough — we are not done"*
- *Framework Completion: "I would pay (or get paid) for that!"*

A successful framework is more than many patterns grouped together. Many patterns together is just a sea of calls — like a large city street at rush hour. There is a pattern of when people stop and go, make turns, speed up, or yield to let someone into traffic. Individual tasks are accomplished, but even if you could step back a bit — there is little to be understood by all the interactions.

- *"Where is everyone going?"*

A framework normally has a complex purpose. We have typically accomplished something of significance or difficulty once we have harnessed a framework to perform a specific goal. Users of frameworks are commonly not alone. Similar accomplishments are achieved by others with similar challenges but varying requirements.

- *"This has gotten many to their target. You just need to supply ..."*

Well designed and popular frameworks can operate at different scale — not just a one-size-fits-all all-of-the-time. This could be for different sized environments or simply for developers to have a workbench to learn with, demonstrate, or develop components for specific areas.

- *"Why does the map have to be actual size?"*

9.2. Framework Characteristics

The following distinguishing features for a framework are listed on Wikipedia.^[1] I will use them to structure some further explanations.

Inversion of Control (IoC)

Unlike a procedural algorithm where our concrete code makes library calls to external components, a framework calls our code to do detailed things at certain points. All the complex but reusable logic has been abstracted into the framework.

- "*Don't call us. We'll call you.*" is a very common phrase to describe inversion of control

Default Behavior

Users of the framework do not have to supply everything. One or more selectable defaults try to do the common, right thing.

- *Remember—the framework developers have solved this before and have harvested the key abstractions and processing from the skeletal remains of previous solutions*

Extensibility

To solve the concrete case, users of the framework must be able to provide specializations that are specific to their problem domain.

- *Framework developers—understanding the problem domain—have pre-identified which abstractions will need to be specialized by users. If they get that wrong, it is a sign of a bad framework.*

Non-modifiable Framework code

A framework has a tangible structure; well-known abstractions that perform well-defined responsibilities. That tangible aspect is visible in each of the concrete solutions and is what makes the product of a framework immediately understandable to other users of the framework.

- "*This is very familiar.*"

[1] "Software framework", Wikipedia

Chapter 10. Framework Enablers

10.1. Dependency Injection

A process to enable Inversion of Control (IoC), whereby objects define their dependencies ^[1] and the manager (the "Container") assembles and connects the objects according to definitions.

The "manager" can be your setup code ("POJO" setup) or in realistic cases a "container" (see later definition)

10.2. POJO

A Plain Old Java Object (POJO) is what the name says it is. It is nothing more than an instantiated Java class.

A POJO normally will address the main purpose of the object and can be missing details or dependencies that give it complete functionality. Those details or dependencies are normally for specialization and extensibility that is considered outside the main purpose of the object.

- *Example: POJO may assume inputs are valid but does not know validation rules.*

10.3. Component

A component is a fully assembled set of code (one or more POJOs) that can perform its duties for its clients. A component will normally have a well-defined interface and a well-defined set of functions it can perform.

A component can have zero or more dependencies on other components, but there should be no further **mandatory** assembly once your client code gains access to it.

10.4. Bean

A generalized term that tends to refer to an object in the range of a POJO to a component that encapsulates something. A supplied "bean" takes care of aspects that we do not need to have knowledge of.

In Spring, the objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and managed by a Spring IoC container. Otherwise, a bean is simply one of many objects in your application. Beans, and the dependencies among them, are reflected in the configuration metadata used by a container. ^[1]

— Spring.io, Introduction to the Spring IoC Container and Beans



You will find that I commonly use the term "component" in the lecture notes—to be a bean that is fully assembled and managed by the container.

10.5. Container

A container is the assembler and manager of components.

Both Docker and Spring are two popular containers that work at two different levels but share the same core responsibility.

10.5.1. Docker Container Definition

- Docker supplies a container that assembles and packages software so that it can be generically executed on remote platforms.

A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. ^[2]

— Docker.com, Use containers to Build Share and Run your applications

10.5.2. Spring Container Definition

- Spring supplies a container that assembles and packages software to run within a JVM.

(The container) is responsible for instantiating, configuring, and assembling the beans. The container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata. The configuration metadata is represented in XML, Java annotations, or Java code. It lets you express the objects that compose your application and the rich interdependencies between those objects. ^[3]

— Spring.io, Container Overview

10.6. Interpose

(Framework/Spring) Containers do more than just configure and assemble simple POJOs. Containers can apply layers of functionality onto beans when wrapping them into components. Examples:

- Perform validation
- Enforce security constraints
- Manage transaction for backend resource
- Perform method in a separate thread

10.6.1. POJO Calls

The following two examples are examples of straight POJO calls. There is no interpose going on here.

In the first example, method `m1()` and `m2()` are in the same class (aka "buddy methods"). Method `m1()` calls sibling buddy method `m2()`. This call will be a straight POJO call. No container is involved between two methods of the same class unless there is a chance for sub-classing.

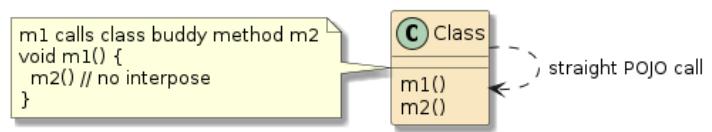


Figure 2. POJO Buddy Call

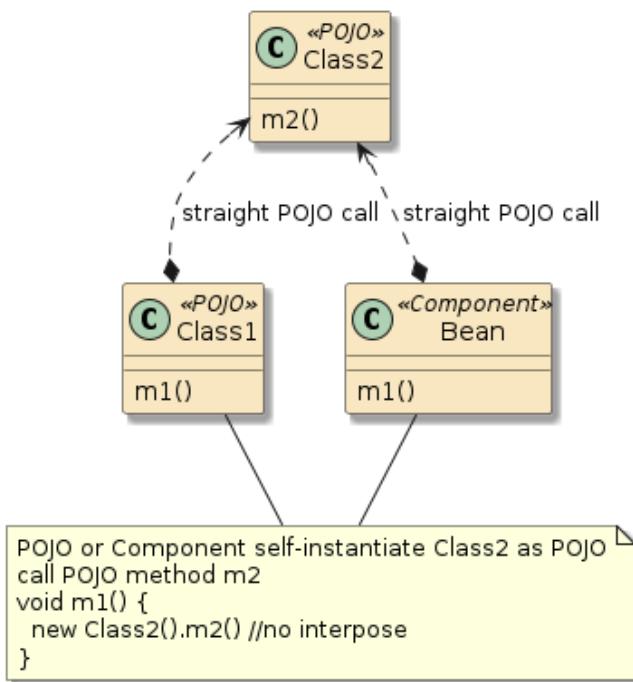


Figure 3. Self-Instantiated POJO Call

In the second example, method `m1()` and `m2()` are in two separate classes. Method `m2()` is inside `Class2`. Method `m1()` instantiates `Class2` and calls method `m2()`. This call will also be a straight POJO call no matter whether `m1()` is a POJO or component because `Class2` was instantiated outside the control of the container.

10.6.2. Container Interpose

In this third example, method `m1()` and method `m2()` are in two separate classes (`Class1` and `Class2`)—but those classes have been defined as beans to the container.

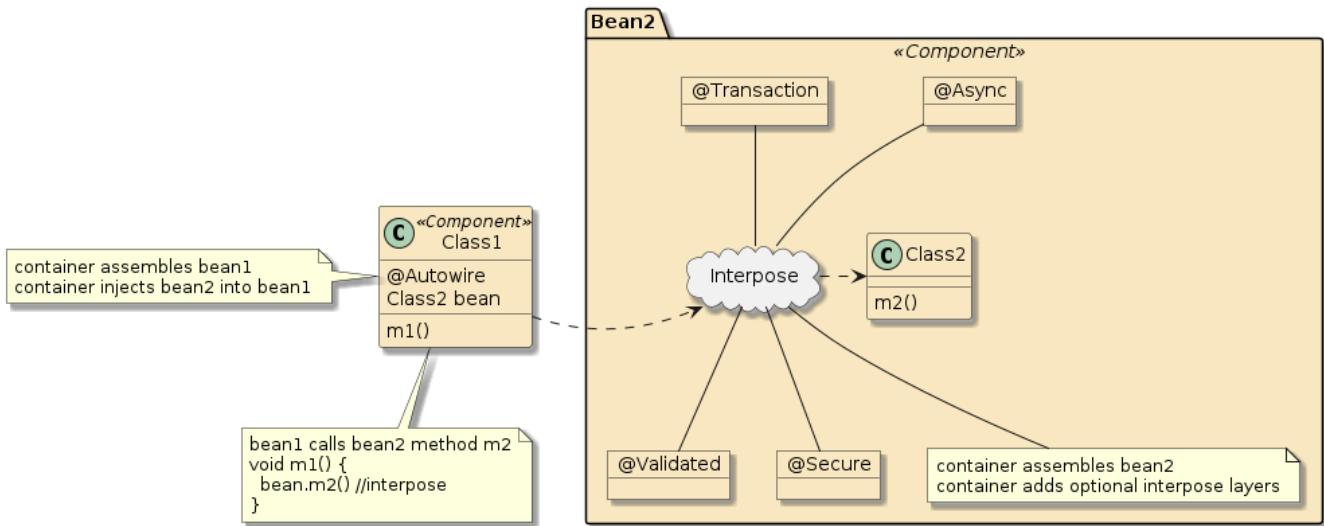


Figure 4. Container Interpose

That means both **Class1** and **Class2** will be instantiated as beans by the container. **Bean2** will be augmented with zero or more layers of functionality—called **interpose**—to implement the full bean component definition. **Bean1** will have the **Bean2** injected to satisfy its **Class2** dependency and be augmented with whatever functionality its is required to complete its bean component definition

This is how features can be added to simple looking POJOs when we make them into beans.

- [1] "Spring Framework Documentation, The IoC Container", Spring.io
- [2] "Use containers to Build, Share and Run your applications", Docker.com
- [3] "The IOC Container, Container Overview", Spring Framework Documentation

Chapter 11. Language Impact on Frameworks

As stated earlier, frameworks provide a template of behavior—allowing for configuration and specialization. Over the years, the ability to configure and to specialize has gone through significant changes with language support.

11.1. XML Configurations

Prior to Java 5, the primary way to identify components was with an XML file. The XML file would identify a bean class provided by the framework user. The bean class would either implement an interface or comply with JavaBean getter/setter conventions.

11.1.1. Inheritance

Early JavaEE EJB defined a set of interfaces that represented things like stateless and stateful sessions and persistent entity classes. End-users would implement the interface to supply specializations for the framework. These interfaces had many callbacks that were commonly not needed but had to be tediously implemented with noop return statements—which produced some code bloat. Java interfaces did not support default implementations at that time.

11.1.2. Java Reflection

Early Spring bean definitions used some interface implementation, but more heavily leveraged compliance to JavaBean setter/getter behavior and Java reflection. Bean classes listed in the XML were scanned for methods that started with "set" or "get" (or anything else specially configured) and would form a call to them using Java reflection. This eliminated much of the need for strict interfaces and noop boilerplate return code.

11.2. Annotations

By the time Java 5 and annotations arrived in 2005 (late 2004), the Java framework worlds were drowning in XML. During that early time, everything was required to be defined. There were no defaults.

Although changes did not seem immediate, JavaEE frameworks like EJB 3.0/JPA 1.0 provided a substantial example for the framework communities in 2006. They introduced "sane" defaults and a primary (XML) and secondary (annotation) override system to give full choice and override of how to configure. Many things just worked right out of the box and only required a minor set of annotations to customize.

Spring went a step further and created a Java Configuration capability to be a 100% replacement for the old XML configurations. XML files were replaced by Java classes. XML bean definitions were replaced by annotated factory methods. Bean construction and injection was replaced by instantiation and setter calls within the factory methods.

Both JavaEE and Spring supported class level annotations for components that were very simple to instantiate and followed standard injection rules.

11.3. Lambdas

Java 8 brought in lambdas and functional processing, which from a strictly syntactical viewpoint is primarily a shorthand for writing an implementation to an interface (or abstract class) with only one abstract method.

You will find many instances in modern libraries where a call will accept a lambda function to implement core business functionality within the scope of the called method. Although—as stated—this is primarily syntactical sugar, it has made method definitions so simple that many more calls take optional lambdas to provide convenient extensions.

Chapter 12. Key Frameworks

In this section I am going to list a limited set of key Java framework highlights. In following the primarily Java path for enterprise frameworks, you will see a remarkable change over the years.

12.1. CGI Scripts

The Common Gateway Interface (CGI) was the cornerstone web framework when Java started coming onto the scene.^[1] CGI was created in 1993 and, for the most part, was a framework for accepting HTTP calls, serving up static content and calling scripts to return dynamic content results.^[2]

The important parts to remember is that CGI was 100% stateless relative to backend resources. Each dynamic script called was a new, heavyweight operating system process and new connection to the database. Java programs were shoehorned into this framework as scripts.

12.2. JavaEE

Jakarta EE, formerly the Java Platform, Enterprise Edition (JavaEE) and Java 2 Platform, Enterprise Edition (J2EE) is a framework that extends the Java Platform, Standard Edition (Java SE) to be an end-to-end Web to database functionality and more.^[3] Focusing only on the web and database portions here, JakartaEE provided a means to invoke dynamic scripts—written in Java—with a process thread and cached database connections.

The initial versions of Jakarta EE aimed big. Everything was a large problem and nothing could be done simply. It was viewed as being overly complex for most users. Spring was formed initially as a means to make J2EE simpler and ended up soon being an independent framework of its own.

J2EE first was released in 1999 and guided by Sun Microsystems. The Servlet portion was likely the most successful portion of the early release. The Enterprise Java Beans (EJB) portion was not realistically usable until JavaEE 5 / post 2006. By then, frameworks like Spring had taken hold of the target community.

In 2010, Sun Microsystems and control of both JavaSE and JavaEE was purchased by Oracle and seemed to progress but on a slow path. By JavaEE 8 in 2017, the framework had become very Spring-like with its POJO-based design. In 2017, Oracle transferred ownership of JavaEE to Jakarta. The JavaEE framework and libraries paused for a few years for naming changes and compatibility releases.^[3]

12.3. Spring

Spring 1.0 was released in 2004 and was an offshoot of a book written by Rod Johnson "Expert One-on-One J2EE Design and Development" that was originally meant to explain how to be successful with J2EE.^[4]

In a nutshell, Rod Johnson and the other designers of Spring thought that rather than starting with a large architecture like J2EE, one should start with a simple bean and scale up from there without

boundaries. Small Spring applications were quickly achieved and gave birth to other frameworks like the Hibernate persistence framework (first released in 2003) which significantly influenced the EJB3/JPA standard. ^[5]

Spring and Spring Boot use many JavaEE(javax)/Jakarta libraries. Spring 5 / Spring Boot 2.7 updated to Jakarta Maven artifacts that still used the `javax` Java package naming at the code level. Spring 6 / Spring Boot 3 updated to Jakarta Maven artifact versions that renamed classes/properties to the `jakarta` package naming.

12.4. Jakarta Persistence API (JPA)

The Jakarta Persistence API (JPA), formerly the Java Persistence API, was developed as a part of the JavaEE community and provided a framework definition for persisting objects in a relational database. JPA fully replaced the original EJB Entity Beans standards of earlier releases. It has an API, provider, and user extensions. ^[6] The main drivers of JPA were EclipseLink (formerly TopLink from Oracle) and Hibernate.

Frameworks should be based on the skeletons of successful implementations



Early EJB Entity Bean standards (< 3) were not thought to have been based on successful implementations. The persistence framework failed to deliver, was modified with each major release, and eventually replaced by something that formed from industry successes.

JPA has been a wildly productive API. It provides simple API access and many extension points for DB/SQL-aware developers to supply more efficient implementations. JPA's primary downside is likely that it allows Java developers to develop persistent objects without thinking of database concerns first. One could hardly blame that on the framework.

12.5. Spring Data

Spring Data is a data access framework centered around a core data object and its primary key—which is very synergistic with Domain-Driven Design (DDD) Aggregate and Repository concepts. ^[7]

- Persistence models like JPA allow relationships to be defined to infinity and beyond.
- In DDD the persisted object has a firm boundary and only IDs are allowed to be expressed when crossing those boundaries.
- These DDD boundary concepts are very consistent with the development of microservices—where large transactional, monoliths are broken down into eventually consistent smaller services.

By limiting the scope of the data object relationships, Spring has been able to automatically define an extensive CRUD (Create, Read, Update, and Delete), query, and extension framework for persisted objects on multiple storage mechanisms.

We will be working with Spring Data JPA and Spring Data Mongo in this class. With the bounding DDD concepts, the two frameworks have an amazing amount of API synergy between them.

12.6. Spring Boot

Spring Boot was first released in 2014. Rather than take the "build anything you want, any way you want" approach in Spring, Spring Boot provides a framework for providing an opinionated view of how to build applications.^[8]

- By adding a dependency, a default implementation is added with "sane" defaults.
- By setting a few properties, defaults are customized to your desired settings.
- By defining a few beans, you can override the default implementations with local choices.

There is no external container in Spring Boot. Everything gets boiled down to an executable JAR and launched by a simple Java main (and a lot of other intelligent code).

Our focus will be on Spring Boot, Spring, and lower-level Spring and external frameworks.

[1] "Write CGI programs in Java", InfoWorld 1997

[2] "Common Gateway Interface", Wikipedia

[3] "Jakarta EE", Wikipedia

[4] "Spring Framework", Wikipedia

[5] "Hibernate (framework)", Wikipedia

[6] "Jakarta Persistence", JPA

[7] "Domain-Driven Design Reference", Eric Evans Domain Language, Inc. 2015

[8] "History of Spring Framework and Spring Boot", Quick Programming Tips

Chapter 13. Summary

In this module we:

- identified the key differences between a library and framework
- identify the purpose for a framework in solving an application solution
- identify the key concepts that enable a framework
- identify specific constructs that have enabled the advance of frameworks
- identify key Java frameworks that have evolved over the years

Pure Java Main Application

copyright © 2024 jim stafford (jim.stafford@jhu.edu)

Chapter 14. Introduction

This material provides an introduction to building a bare bones Java application using a single, simple Java class, packaging that in a Java ARchive (JAR), and executing it two ways:

- as a class in the classpath
- as the Main-Class of a JAR

14.1. Goals

The student will learn:

- foundational build concepts for simple, pure-Java solution

14.2. Objectives

At the conclusion of this lecture and related exercises, the student will be able to:

1. create source code for an executable Java class
2. add that Java class to a Maven module
3. build the module using a Maven pom.xml
4. execute the application using a classpath
5. configure the application as an executable JAR
6. execute an application packaged as an executable JAR

Chapter 15. Simple Java Class with a Main

Our simple Java application starts with a public class with a static main() method that optionally accepts command-line arguments from the caller

```
package info.ejava.examples.app.build.javamain;

import java.util.List;

public class SimpleMainApp { ①
    public static void main(String...args) { ② ③
        System.out.println("Hello " + List.of(args));
    }
}
```

① public class

② implements a static main() method

③ optionally accepts arguments

Chapter 16. Project Source Tree

This class is placed within a module source tree in the `src/main/java` directory below a set of additional directories (`info/ejava/examples/app/build/javamain`) that match the Java package name of the class (`info.ejava.examples.app.build.javamain`)

```
|-- pom.xml ①
`-- src
  |-- main ②
  |   |-- java
  |   |   '-- info
  |   |       '-- ejava
  |   |           '-- examples
  |   |               '-- app
  |   |                   '-- build
  |   |                       '-- javamain
  |   |                           '-- SimpleMainApp.java
  |   '-- resources ③
  '-- test ④
    |-- java
    '-- resources
```

① `pom.xml` will define our project artifact and how to build it

② `src/main` will contain the pre-built, source form of our artifacts that will be part of our primary JAR output for the module

③ `src/main/resources` is commonly used for property files or other resource files read in during the program execution

④ `src/test` is will contain the pre-built, source form of our test artifacts. These will not be part of the primary JAR output for the module

Chapter 17. Building the Java Archive (JAR) with Maven

In setting up the build within Maven, I am going to limit the focus to just compiling our simple Java class and packaging that into a standard Java JAR.

17.1. Add Core pom.xml Document

Add the core document with required GAV information (`groupId`, `artifactId`, `version`) to the `pom.xml` file at the root of the module tree. Packaging is also required but will have a default of `jar` if not supplied.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>info.ejava.examples.app</groupId> ①
    <artifactId>java-app-example</artifactId> ②
    <version>6.1.0-SNAPSHOT</version> ③
    <packaging>jar</packaging> ④
<project>
```

① `groupId`

② `artifactId`

③ `version`

④ `packaging`



Module directory should be the same name/spelling as `artifactId` to align with default directory naming patterns used by plugins.



Packaging specification is optional in this case. The default packaging is `jar`

17.2. Add Optional Elements to pom.xml

- `name`

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
```

```

<groupId>info.ejava.examples.app</groupId>
<artifactId>java-app-example</artifactId>
<version>6.1.0-SNAPSHOT</version>
<packaging>jar</packaging>

<name>App::Build::Java Main Example</name> ①
<project>

```

① name appears in Maven build output but not required

17.3. Define Plugin Versions

Define plugin versions so the module can be deterministically built in multiple environments

- Each version of Maven has a set of default plugins and plugin versions
- Each plugin version may or may not have a set of defaults (e.g., not Java 17) that are compatible with our module

```

<properties>
    <java.target.version>17</java.target.version>
    <maven-compiler-plugin.version>3.13.0</maven-compiler-plugin.version>
    <maven-jar-plugin.version>3.4.2</maven-jar-plugin.version>
</properties>

<pluginManagement>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>${maven-compiler-plugin.version}</version>
            <configuration>
                <release>${java.target.version}</release>
            </configuration>
        </plugin>
    </plugins>
</pluginManagement>

```

The `jar` packaging will automatically activate the `maven-compiler-plugin` and `maven-jar-plugin`. Our definition above identifies the version of the plugin to be used (if used) and any desired configuration of the plugin(s).

17.4. pluginManagement vs. plugins

- Use `pluginManagement` to define a plugin if it activated in the module build
 - useful to promote consistency in multi-module builds
 - commonly seen in parent modules

- Use `plugins` to declare that a plugin be active in the module build
 - ideally only used by child modules
 - our child module indirectly activated several plugins by using the `jar` packaging type

Chapter 18. Build the Module

Maven modules are commonly built with the following commands/ [phases](#)

- `clean` removes previously built artifacts
- `package` creates primary artifact(s) (e.g., JAR)
 - processes main and test resources
 - compiles main and test classes
 - runs unit tests
 - builds the archive

```
[INFO] Scanning for projects...
[INFO]
[INFO] -----< info.ejava.examples.app:java-app-example >-----
[INFO] Building App::Build::Java App Example 6.1.0-SNAPSHOT
[INFO]   from pom.xml
[INFO] -----[ jar ]-----
[INFO]
[INFO] --- maven-clean-plugin:3.4.0:clean (default-clean) @ java-app-example ---
[INFO] Deleting .../java-app-example/target
[INFO]
[INFO] --- maven-resources-plugin:3.3.1:resources (default-resources) @ java-app-
example ---
[INFO] Copying 0 resource from src/main/resources to target/classes
[INFO]
[INFO] --- maven-compiler-plugin:3.13.0:compile (default-compile) @ java-app-example
---
[INFO] Recompiling the module because of changed source code.
[INFO] Compiling 1 source file with javac [debug parameters release 17] to
target/classes
[INFO]
[INFO] --- maven-resources-plugin:3.3.1:testResources (default-testResources) @ java-
app-example ---
[INFO] Copying 0 resource from src/test/resources to target/test-classes
[INFO]
[INFO] --- maven-compiler-plugin:3.13.0:testCompile (default-testCompile) @ java-app-
example ---
[INFO] Recompiling the module because of changed dependency.
[INFO]
[INFO] --- maven-surefire-plugin:3.3.1:test (default-test) @ java-app-example ---
[INFO]
[INFO] --- maven-jar-plugin:3.4.2:jar (default-jar) @ java-app-example ---
[INFO] Building jar: .../java-app-example/target/java-app-example-6.1.0-SNAPSHOT.jar
[INFO]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
```

[INFO] Total time: 1.783 s

Chapter 19. Project Build Tree

The produced build tree from `mvn clean package` contains the following key artifacts (and more)

```
|-- pom.xml
|-- src
`-- target
    |-- classes ①
    |   '-- info
    |       '-- ejava
    |           '-- examples
    |               '-- app
    |                   '-- build
    |                       '-- javemain
    |                           '-- SimpleMainApp.class
...
|   |-- java-app-example-6.1.0-SNAPSHOT.jar ②
...
`-- test-classes ③
```

① `target/classes` for built artifacts from `src/main`

② primary artifact(s) (e.g., Java Archive (JAR))

③ `target/test-classes` for built artifacts from `src/test`

Chapter 20. Resulting Java Archive (JAR)

Maven adds a few extra files to the META-INF directory that we can ignore. The key files we want to focus on are:

- `SimpleMainApp.class` is the compiled version of our application
- [META-INF/MANIFEST.MF](<https://docs.oracle.com/javase/tutorial/deployment/jar/manifestindex.html>) contains properties relevant to the archive

```
$ jar tf target/java-app-example-*-SNAPSHOT.jar | egrep -v "/" | sort
META-INF/MANIFEST.MF
META-INF/maven/info.ejava.examples.app/java-app-example/pom.properties
META-INF/maven/info.ejava.examples.app/java-app-example/pom.xml
info/ejava/examples/app/build/javamain/SimpleMainApp.class
```



- `jar tf` lists the contents of the JAR
- `egrep` is being used to exclude non-files (i.e., directores) that end with "/"
- `sort` performs an ordering of the output
- `|` pipe character sends the stdout of previous command to the stdin of the next command

Chapter 21. Execute the Application

The application is executed by

- invoking the `java` command
- adding the JAR file (and any other dependencies) to the classpath
- specifying the fully qualified class name of the class that contains our `main()` method

Example with no arguments

```
$ java -cp target/java-app-example-*-SNAPSHOT.jar  
info.ejava.examples.app.build.javamain.SimpleMainApp
```

Output:

```
Hello []
```

Example with arguments

```
$ java -cp target/java-app-example-*-SNAPSHOT.jar  
info.ejava.examples.app.build.javamain.SimpleMainApp arg1 arg2 "arg3 and 4"
```

Output:

```
Hello [arg1, arg2, arg3 and 4]
```

- example passed three (3) arguments separated by spaces
 - third argument (`arg3 and arg4`) used quotes around the entire string to escape spaces and have them included in the single parameter

Chapter 22. Configure Application as an Executable JAR

To execute a specific Java class within a classpath is conceptually simple. However, there is a lot more to know than we need to when there may be only a single entry point. In the following sections we will assign a default Main-Class by using the [MANIFEST.MF properties](#)

22.1. Add Main-Class property to MANIFEST.MF

```
$ unzip -qc target/java-app-example-*-SNAPSHOT.jar META-INF/MANIFEST.MF  
  
Manifest-Version: 1.0  
Created-By: Maven JAR Plugin 3.4.2  
Build-Jdk-Spec: 17  
Main-Class: info.ejava.examples.app.build.javamain.SimpleMainApp
```

22.2. Automate Additions to MANIFEST.MF using Maven

One way to surgically add that property is thru the [maven-jar-plugin](#)

```
<plugin>  
  <groupId>org.apache.maven.plugins</groupId>  
  <artifactId>maven-jar-plugin</artifactId>  
  <version>${maven-jar-plugin.version}</version>  
  <configuration>  
    <archive>  
      <manifest>  
        <mainClass>  
info.ejava.examples.app.build.javamain.SimpleMainApp</mainClass>  
        </manifest>  
      </archive>  
    </configuration>  
  </plugin>
```



This is a very specific plugin configuration that would only apply to a specific child module. Therefore, we would place this in a [plugins](#) declaration versus a [pluginsManagement](#) definition.

Chapter 23. Execute the JAR versus just adding to classpath

The executable JAR is executed by

- invoking the `java` command
- adding the `-jar` option
- adding the JAR file (and any other dependencies) to the classpath

Example with no arguments

```
$ java -jar target/java-app-example-*-SNAPSHOT.jar
```

Output:

```
Hello []
```

Example with arguments

```
$ java -jar target/java-app-example-*-SNAPSHOT.jar one two "three and four"
```

Output:

```
Hello [one, two, three and four]
```

- example passed three (3) arguments separated by spaces
 - third argument (`three and four`) used quotes around the entire string to escape spaces and have them included in the single parameter

Chapter 24. Configure pom.xml to Test

At this point we are ready to create an automated execution of our JAR as a part of the build. We have to do that after the `packaging` phase and will leverage the `integration-test` Maven phase

```
<build>
  ...
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-antrun-plugin</artifactId> ①
    <executions>
      <execution>
        <id>execute-jar</id>
        <phase>integration-test</phase> ④
        <goals>
          <goal>run</goal>
        </goals>
        <configuration>
          <tasks>
            <java fork="true" classname=
"info.ejava.examples.app.build.javamain.SimpleMainApp"> ②
              <classpath>
                <pathElement path=
"${project.build.directory}/${project.build.finalName}.jar"/>
              </classpath>
              <arg value="Ant-supplied java -cp"/>
              <arg value="Command Line"/>
              <arg value="args"/>
            </java>

            <java fork="true"
                  jar=
"${project.build.directory}/${project.build.finalName}.jar"> ③
              <arg value="Ant-supplied java -jar"/>
              <arg value="Command Line"/>
              <arg value="args"/>
            </java>
          </tasks>
        </configuration>
      </execution>
    </executions>
  </plugin>
</plugins>
</build>
```

① Using the `maven-ant-run` plugin to execute Ant task

② Using the `java` Ant task to execute shell `java -cp` command line

③ Using the `java` Ant task to execute shell `java -jar` command line

④ Running the plugin during the `integration-phase`

- Order
 - 1. `package`
 - 2. `pre-integration`
 - 3. `integration-test`
 - 4. `post-integration`
 - 5. `verify`

24.1. Execute JAR as part of the build

```
$ mvn clean verify
[INFO] Scanning for projects...
[INFO]
[INFO] -----< info.ejava.examples.app:java-app-example >-----
...
[INFO] --- maven-jar-plugin:3.2.2:jar (default-jar) @ java-app-example -①
[INFO] Building jar: .../java-app-example/target/java-app-example-6.1.0-SNAPSHOT.jar
[INFO]
...
[INFO] --- maven-antrun-plugin:3.1.0:run (execute-jar) @ java-app-example ---
[INFO] Executing tasks ②
[INFO]      [java] Hello [Ant-supplied java -cp, Command Line, args]
[INFO]      [java] Hello [Ant-supplied java -jar, Command Line, args]
[INFO] Executed tasks
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
```



```
[INFO] --- maven-jar-plugin:3.4.2:jar (default-jar) @ java-app-example -①
[INFO] Building jar: .../java-app-example/target/java-app-example-6.1.0-SNAPSHOT.jar
[INFO]
...
[INFO] --- maven-antrun-plugin:3.1.0:run (execute-jar) @ java-app-example ---
[INFO] Executing tasks ②
[INFO]      [java] Hello [Ant-supplied java -cp, Command Line, args]
[INFO]      [java] Hello [Ant-supplied java -jar, Command Line, args]
[INFO] Executed tasks
[INFO] -----
[INFO] BUILD SUCCESS
```

① Our plugin is executing

② Our application was executed and the results displayed

Chapter 25. Summary

1. The JVM will execute the static `main()` method of the class specified in the `java` command
2. The class must be in the JVM classpath
3. Maven can be used to build a JAR with classes
4. A JAR can be the subject of a java execution
5. The Java `META-INF/MANIFEST.MF Main-Class` property within the target JAR can express the class with the `main()` method to execute
6. The `maven-jar-plugin` can be used to add properties to the `META-INF/MANIFEST.MF` file
7. A Maven build can be configured to execute a JAR

Simple Spring Boot Application

copyright © 2024 jim stafford (jim.stafford@jhu.edu)

Chapter 26. Introduction

This material makes the transition from a creating and executing a simple Java main application to a Spring Boot application.

26.1. Goals

The student will learn:

- foundational build concepts for simple, Spring Boot Application

26.2. Objectives

At the conclusion of this lecture and related exercises, the student will be able to:

1. extend the standard Maven `jar` module packaging type to include core Spring Boot dependencies
2. construct a basic Spring Boot application
3. build and execute an executable Spring Boot JAR
4. define a simple Spring component and inject that into the Spring Boot application

Chapter 27. Spring Boot Maven Dependencies

Spring Boot provides a [spring-boot-starter-parent](#) ([gradle source](#), [pom.xml](#)) pom that can be used as a parent pom for our Spring Boot modules.^[1] This defines version information for dependencies and plugins for building Spring Boot artifacts—along with an opinionated view of how the module should be built.

[spring-boot-starter-parent](#) inherits from a [spring-boot-dependencies](#) ([gradle source](#), [pom.xml](#)) pom that provides a definition of artifact versions without an opinionated view of how the module is built. This pom can be imported by modules that already inherit from a local Maven parent—which would be common. This is the demonstrated approach we will take here. We will also include demonstration of how the build constructs are commonly spread across parent and local poms.



Spring Boot has converted over to gradle and posts a pom version of the gradle artifact to [Maven central repository](#) as a part of their build process.

[1] [Spring Boot and Build Systems](#), Pivotal

Chapter 28. Parent POM

We are likely to create multiple Spring Boot modules and would be well-advised to begin by creating a local parent pom construct to house the common passive definitions. By passive definitions (versus active declarations), I mean definitions for the child poms to use if needed versus mandated declarations for each child module. For example, a parent pom may define the JDBC driver to use when needed but not all child modules will need a JDBC driver nor a database for that matter. In that case, we do not want the parent pom to actively declare a dependency. We just want the parent to passively define the dependency that the child can optionally choose to actively declare. This construct promotes consistency among all the modules.

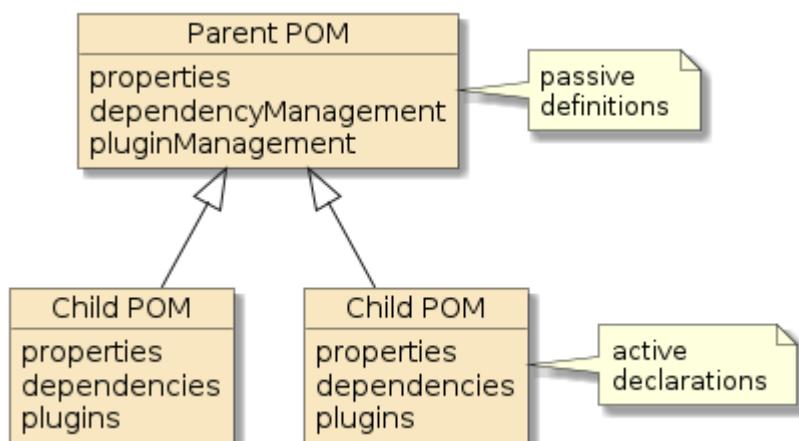


Figure 5. Parent/Child Pom Relationship and Responsibilities



"Root"/parent poms should define dependencies and plugins for consistent re-use among child poms and use dependencyManagement and pluginManagement elements to do so.



"Child"/concrete/leaf poms declare dependencies and plugins to be used when building that module and try to keep dependencies to a minimum.



"Prototype" poms are a blend of root and child pom concepts. They are a nearly-concrete, parent pom that can be extended by child poms but actively declare a select set of dependencies and plugins to allow child poms to be as terse as possible.

28.1. Define Version for Spring Boot artifacts

Define the version for Spring Boot artifacts to use. I am using a technique below of defining the value in a property so that it is easy to locate and change as well as re-use elsewhere if necessary.

Explicit Property Definition

```
# Place this declaration in an inherited parent pom
<properties>
    <springboot.version>3.3.2</springboot.version> ①
```

```
</properties>
```

- ① default value has been declared in imported `ejava-build-bom`



Property values can be overruled at build time by supplying a system property on the command line "-D(name)=(value)"

28.2. Import springboot-dependencies-plugin

Import `springboot-dependencies-plugin`. This will define dependencyManagement for us for many artifacts that are relevant to our Spring Boot development.

```
# Place this declaration in an inherited parent pom
<dependencyManagement> ①
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-dependencies</artifactId>
      <version>${springboot.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

- ① import is within `examples-root` for class examples, which is a grandparent of this example

Chapter 29. Local Child/Leaf Module POM

The local child module pom.xml is where the module is physically built. Although Maven modules can have multiple levels of inheritance—where each level is a child of their parent—the child module I am referring to here is the leaf module where the artifacts are meant to be really built. Everything defined above it is primarily used as a common definition (through dependencyManagement and pluginManagement) to simplify the child pom.xml and to promote consistency among sibling modules. It is the job of the leaf module to activate these definitions that are appropriate for the type of module being built.

29.1. Declare pom inheritance in the child pom.xml

Declare pom inheritance in the child pom.xml to pull in definitions from parent pom.xml.

```
# Place this declaration in the child/leaf pom building the JAR archive
<parent>
    <groupId>(parent groupId)</groupId>
    <artifactId>(parent artifactId)</artifactId>
    <version>(parent version)</version>
</parent>
```

The following diagram shows the parent/child relationship between the `springboot-app-example` and the `class-example-root` pom and the parent's relationships.

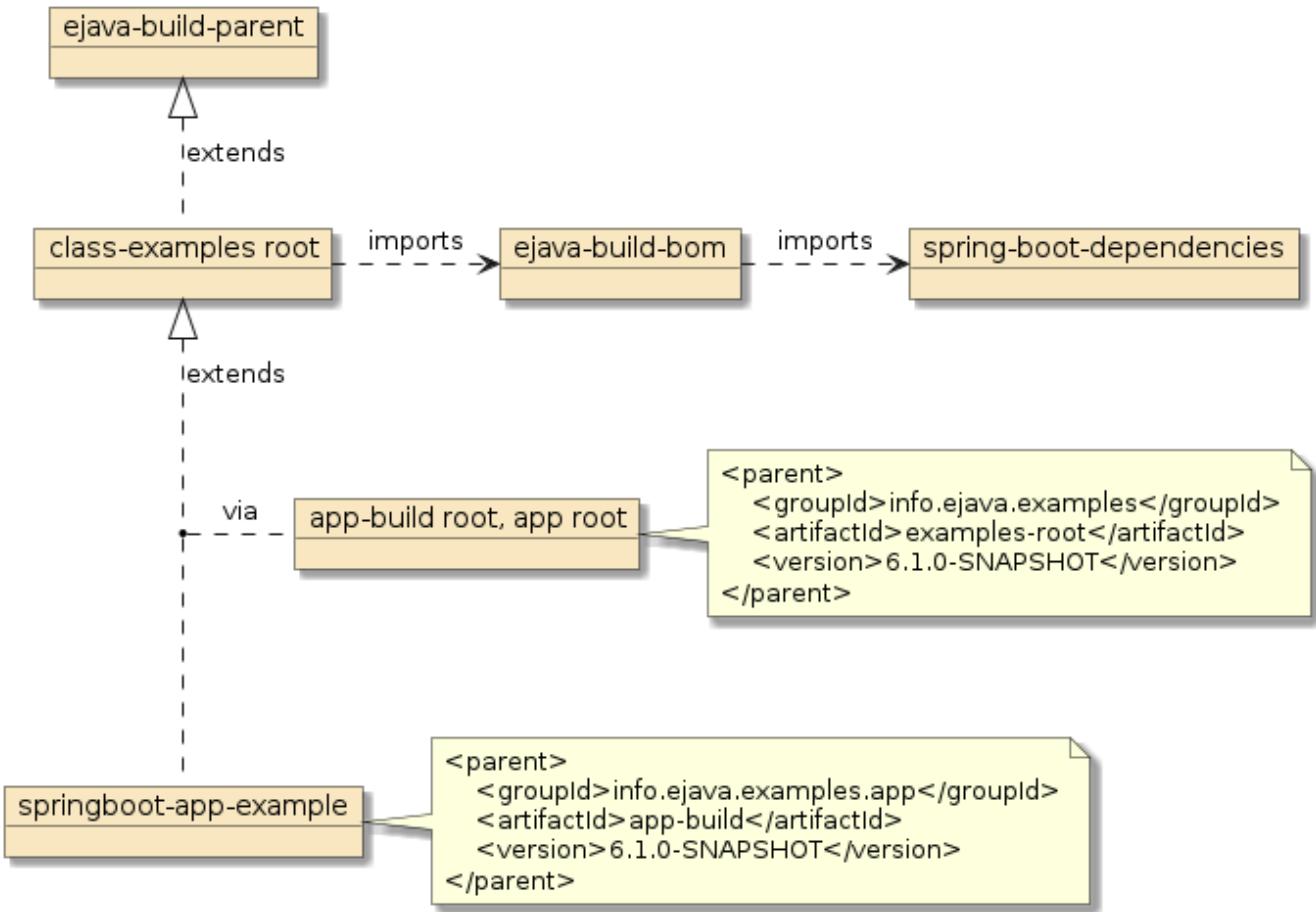


Figure 6. *SpringBoot App Example POM Tree*

29.2. Declare dependency on artifacts used

Realize the parent definition of the `spring-boot-starter` dependency by declaring it within the child dependencies section. For where we are in this introduction, only the above dependency will be necessary. The imported `spring-boot-dependencies` will take care of declaring the version#

```
# Place this declaration in the child/leaf pom building the JAR archive
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
    <!--version --> ①
  </dependency>
</dependencies>
```

① parent has defined (using `import` in this case) the version for all children to consistently use

The figure below shows the parent poms being the source of the passive dependency definitions and the child being the source of the active dependency declarations.

- the parent is responsible for defining the version# for dependencies used
- the child is responsible for declaring what dependencies are needed and adopts the parent version definition

An upgrade to a future dependency version should not require a change of a child module declaration if this pattern is followed.

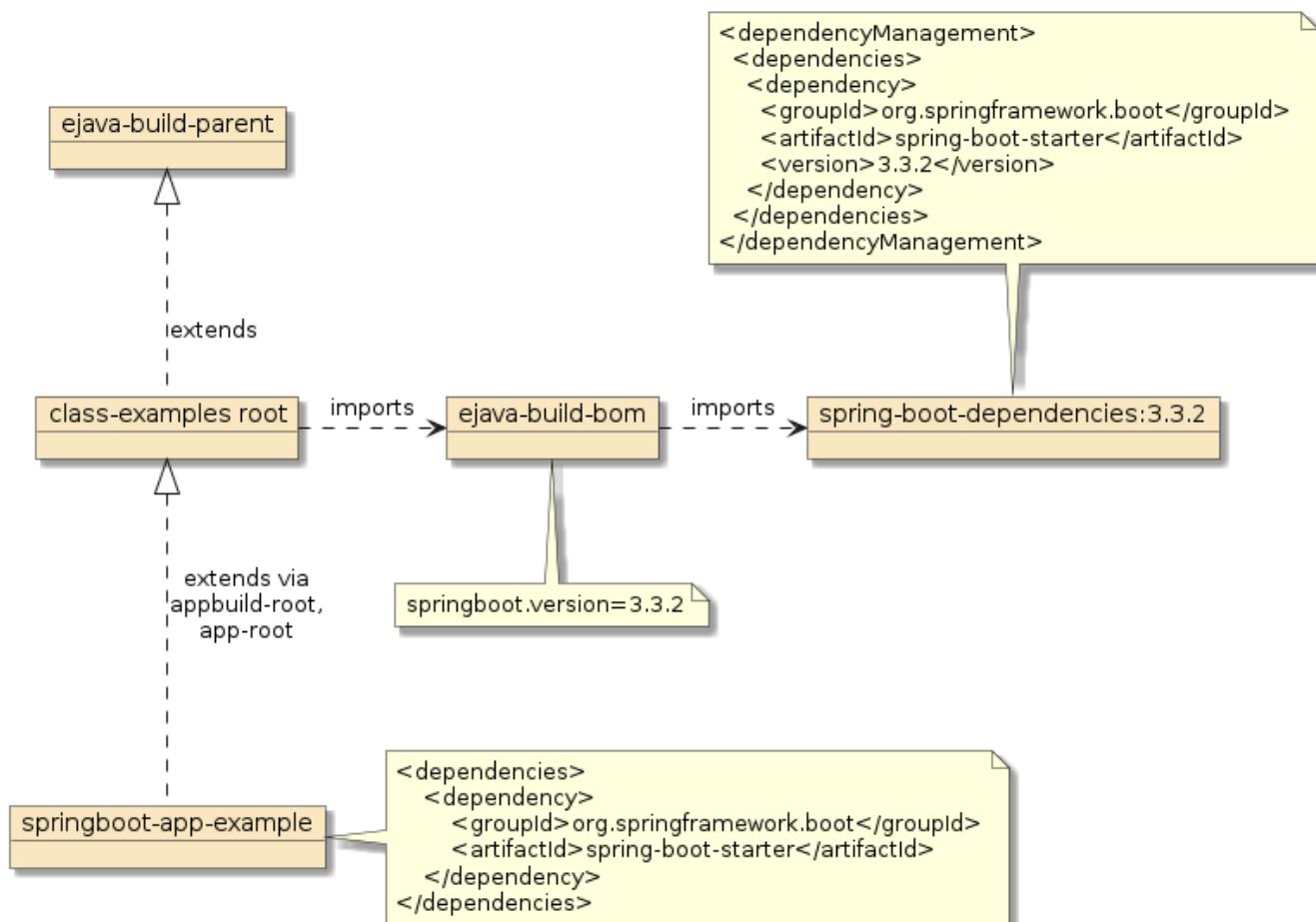


Figure 7. Class Examples dependencyManagement

Chapter 30. Simple Spring Boot Application Java Class

With the necessary dependencies added to our build classpath, we now have enough to begin defining a simple Spring Boot Application.

```
package info.ejava.springboot.examples.app.build.springboot;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication ③
public class SpringBootApp {
    public static void main(String... args) { ①
        System.out.println("Running SpringApplication");

        SpringApplication.run(SpringBootApp.class, args); ②
        System.out.println("Done SpringApplication");
    }
}
```

① Define a class with a static main() method

② Initiate Spring application bootstrap by invoking `SpringApplication.run()` and passing a) application class and b) args passed into main()

③ Annotate the class with `@SpringBootApplication`



Startup can, of course be customized (e.g., change the printed banner, registering event listeners)

30.1. Module Source Tree

The source tree will look similar to our previous Java main example.

```
|-- pom.xml
`-- src
  |-- main
  |   |-- java
  |   |   '-- info
  |   |       '-- ejava
  |   |           '-- examples
  |   |               '-- app
  |   |                   '-- build
  |   |                       '-- springboot
  |   |                           '-- SpringBootApp.java
  |-- resources
```

```
'-- test
  |-- java
  '-- resources
```

30.2. @SpringBootApplication Aggregate Annotation

```
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SpringBootApp {
```

The `@SpringBootApplication` annotation is a compound class-level annotation aggregating the following annotations.

- `@ComponentScan` - legacy Spring annotation that configures component scanning to include or exclude looking through various packages for classes with component annotations
 - By default, scanning will start with the package declaring the annotation and work its way down from there
- `@SpringBootConfiguration` - like legacy Spring `@Configuration` annotation, it signifies the class can provide configuration information. Unlike `@Configuration`, there can be only one `@SpringBootConfiguration` per application, and it is normally supplied by `@SpringBootApplication` except in some integration tests.
 - Classes annotated with `@Configuration` contain factory `@Bean` definitions.
- `@EnableAutoConfiguration` - Allows Spring to perform autoconfiguration based on the classpath, beans defined by the application, and property settings.



The class annotated with `@SpringBootApplication` is commonly located in a Java package that is above all other Java packages containing components for the application.

Chapter 31. Spring Boot Executable JAR

At this point we can likely execute the Spring Boot Application within the IDE but instead, lets go back to the pom and construct a JAR file to be able to execute the application from the command line.

31.1. Building the Spring Boot Executable JAR

We saw earlier how we could build a standard executable JAR using the [maven-jar-plugin](#). However, there were some limitations to that approach—especially the fact that a standard Java JAR cannot house dependencies to form a self-contained classpath and Spring Boot will need additional JARs to complete the application bootstrap. Spring Boot uses a custom executable JAR format that can be built with the aid of the [spring-boot-maven-plugin](#). Let's extend our pom.xml file to enhance the standard JAR to be a Spring Boot executable JAR.

31.1.1. Declare spring-boot-maven-plugin

The following snippet shows the configuration for a [spring-boot-maven-plugin](#) that defines a default execution to build the Spring Boot executable JAR for all child modules that declare using it. In addition to building the Spring Boot executable JAR, we are setting up a standard in the parent for all children to have their follow-on JAR classified separately as a `bootexec`. `classifier` is a core Maven construct and is meant to label sibling artifacts to the original Java JAR for the module. Other types of `classifiers` are `source`, `schema`, `javadoc`, etc. `bootexec` is a value we made up.



`bootexec` is a value we made up.

By default, the `repackage` goal would have replaced the Java JAR with the Spring Boot executable JAR. That would have left an ambiguous JAR artifact in the repository—we would not easily know its JAR type. This will help eliminate dependency errors during the semester when we layer `N+1` assignments on top of layer `N`. Only standard Java JARs can be used in classpath dependencies.

spring-boot-maven-plugin with classifier

```
<properties>
    <spring-boot.classifier>bootexec</spring-boot.classifier>
</properties>
...
<build>
    <pluginManagement>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <classifier>${spring-boot.classifier}</classifier> ④
                </configuration>
                <executions>
                    <execution>
```

```

<id>build-app</id> ①
<phase>package</phase> ②
<goals>
    <goal>repackage</goal> ③
</goals>
</execution>
</executions>
</plugin>
...
</plugins>
</pluginManagement>
</build>

```

- ① `id` used to describe execution and required when having more than one
- ② `phase` identifies the maven goal in which this plugin runs
- ③ `repackage` identifies the goal to execute within the `spring-boot-maven-plugin`
- ④ adds a `-bootexec` to the executable JAR's name

We can do much more with the `spring-boot-maven-plugin` on a per-module basis (e.g., run the application from within Maven). We are just starting at construction at this point.

31.1.2. Build the JAR

```

$ mvn clean package

[INFO] Scanning for projects...
...
[INFO] --- maven-jar-plugin:3.4.2:jar (default-jar) @ springboot-app-example ---
[INFO] Building jar: .../target/springboot-app-example-6.1.0-SNAPSHOT.jar ①
[INFO]

[INFO] --- spring-boot-maven-plugin:3.3.2:repackage (build-app) @ springboot-app-
example ---
[INFO] Attaching repackaged archive .../target/springboot-app-example-6.1.0-SNAPSHOT-
bootexec.jar with classifier bootexec ②

```

- ① standard Java JAR is built by the `maven-jar-plugin`
- ② standard Java JAR is augmented by the `spring-boot-maven-plugin`

31.2. Java MANIFEST.MF properties

The `spring-boot-maven-plugin` augmented the standard JAR by adding a few properties to the `MANIFEST.MF` file

```

$ unzip -qc target/springboot-app-example-6.1.0-SNAPSHOT-bootexec.jar META-
INF/MANIFEST.MF
Manifest-Version: 1.0

```

```
Created-By: Maven JAR Plugin 3.2.2
Build-Jdk-Spec: 17
Main-Class: org.springframework.boot.loader.launch.JarLauncher ①
Start-Class: info.ejava.examples.app.build.springboot.SpringBootApp ②
Spring-Boot-Version: 3.3.2
Spring-Boot-Classes: BOOT-INF/classes/
Spring-Boot-Lib: BOOT-INF/lib/
Spring-Boot-Classpath-Index: BOOT-INF/classpath.idx
Spring-Boot-Layers-Index: BOOT-INF/layers.idx
```

① **Main-Class** was set to a Spring Boot launcher

② **Start-Class** was set to the class we defined with `@SpringBootApplication`

31.3. JAR size

Notice that the size of the Spring Boot executable JAR is significantly larger than the original standard JAR.

```
$ ls -lh target/*jar* | grep -v sources | cut -d\ -f9-99
10M Aug 28 15:19 target/springboot-app-example-6.1.0-SNAPSHOT-bootexec.jar ②
4.3K Aug 28 15:19 target/springboot-app-example-6.1.0-SNAPSHOT.jar ①
```

① The original Java JAR with Spring Boot annotations was 4.3KB

② The Spring Boot JAR is 10MB

31.4. JAR Contents

Ref: [spring.io Appendix E. The Executable Jar Format](#)

Unlike WARs, a standard Java JAR does not provide a standard way to embed dependency JARs. Common approaches to embed dependencies within a single JAR include a "shaded" JAR where all dependency JAR are unwound and packaged as a single "uber" JAR

- positives
 - works
 - follows standard Java JAR constructs
- negatives
 - obscures contents of the application
 - problem if multiple source JARs use files with same path/name

Spring Boot creates a custom WAR-like structure

```
BOOT-INF/classes/info/ejava/examples/app/build/springboot/AppCommand.class
BOOT-INF/classes/info/ejava/examples/app/build/springboot/SpringBootApp.class ③
BOOT-INF/lib/javax.annotation-api-2.1.1.jar ②
```

```
BOOT-INF/lib/spring-boot-3.3.2.jar
BOOT-INF/lib/spring-context-6.1.11.jar
BOOT-INF/lib/spring-beans-6.1.11.jar
BOOT-INF/lib/spring-core-6.1.11.jar
...
META-INF/MANIFEST.MF
META-INF/maven/info.ejava.examples.app/springboot-app-example/pom.properties
META-INF/maven/info.ejava.examples.app/springboot-app-example/pom.xml
org/springframework/boot/loader/launch/ExecutableArchiveLauncher.class ①
org/springframework/boot/loader/launch/JarLauncher.class
...
org/springframework/boot/loader/util/SystemPropertyUtils.class
```

- ① Spring Boot loader classes hosted at the root /
 - ② Local application classes hosted in /BOOT-INF/classes
 - ③ Dependency JARs hosted in /BOOT-INF/lib

Spring Boot can also use a standard WAR structure—to be deployed to a web server.

- 99% of it is a standard WAR
 - /WEB-INF/classes
 - /WEB-INF/lib
 - Spring Boot loader classes hosted at the root /
 - Special directory for dependencies only used for non-container deployment
 - /WEB-INF/lib-provided



31.5. Execute Command Line

```
springboot-app-example$ java -jar target/springboot-app-example-6.1.0-SNAPSHOT-bootexec.jar ①
Running SpringApplication ②
```

```
2019-12-04 09:01:03.014 INFO 1287 --- [main] i.e.e.a.build.springboot.SpringBootApp:  
\  
Starting SpringBootApp on Jamess-MBP with PID 1287 (.../springboot-app-  
example/target/springboot-app-example-6.1.0-SNAPSHOT.jar \
```

```
started by jim in .../springboot-app-example)
2019-12-04 09:01:03.017  INFO 1287 --- [main] i.e.e.a.build.springboot.SpringBootApp:
 \
  No active profile set, falling back to default profiles: default
2019-12-04 09:01:03.416  INFO 1287 --- [main] i.e.e.a.build.springboot.SpringBootApp:
 \
 Started SpringBootApp in 0.745 seconds (JVM running for 1.13)
Done SpringApplication ④
```

- ① Execute the JAR using the `java -jar` command
- ② Main executes and passes control to SpringApplication
- ③ Spring Boot bootstrap is started
- ④ SpringApplication terminates and returns control to our `main()`

Chapter 32. Add a Component to Output Message and Args

We have a lot of capability embedded into our current Spring Boot executable JAR that is there to bootstrap the application by looking around for components to activate. Let's explore this capability with a simple class that will take over the responsibility for the output of a message with the arguments to the program.

We want this class found by Spring's application startup processing, so we will:

```
// AppCommand.java
package info.ejava.examples.app.build.springboot; ②

import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;
import java.util.List;

@Component ①
public class AppCommand implements CommandLineRunner {
    public void run(String... args) throws Exception {
        System.out.println("Component code says Hello " + List.of(args));
    }
}
```

① Add a @Component annotation on the class

② Place the class in a Java package configured to be scanned

32.1. @Component Annotation

```
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

@Component
public class AppCommand implements CommandLineRunner {
```

Classes can be configured to have their instances managed by Spring. Class annotations can be used to express the purpose of a class and to trigger Spring into managing them in specific ways. The most generic form of component annotation is `@Component`. Others will include `@Repository`, `@Controller`, etc. Classes directly annotated with a `@Component` (or other annotation) indicates that Spring can instantiate instances of this class with no additional assistance from a `@Bean` factory.

32.2. Interface: CommandLineRunner

```
import org.springframework.boot.CommandLineRunner;
```

```

import org.springframework.stereotype.Component;
@Component
public class AppCommand implements CommandLineRunner {
    public void run(String... args) throws Exception {
    }
}

```

- Components implementing `CommandLineRunner` interface get called after application initialization
- Program arguments are passed to the `run()` method
- Can be used to perform one-time initialization at start-up
- Alternative Interface: `ApplicationRunner`
 - Components implementing `ApplicationRunner` are also called after application initialization
 - Program arguments are passed to its `run()` method have been wrapped in `ApplicationArguments` convenience class



Component startup can be ordered with the `@Ordered` Annotation.

32.3. @ComponentScan Tree

By default, the `@SpringBootApplication` annotation configured Spring to look at and below the Java package for our `SpringBootApp` class. I chose to place this component class in the same Java package as the application class

```

@SpringBootApplication
// @ComponentScan
// @SpringBootConfiguration
// @EnableAutoConfiguration
public class SpringBootApp {
}

```

```

src/main/java
`-- info
  '-- ejava
    '-- springboot
      '-- examples
        '-- app
          |-- AppCommand.java
          '-- SpringBootApp.java

```

Chapter 33. Running the Spring Boot Application

```
$ java -jar target/springboot-app-example-6.1.0-SNAPSHOT-bootexec.jar
```

Running SpringApplication ①

```
2019-09-06 15:56:45.666 INFO 11480 --- [           main]
i.e.s.examples.app.SpringBootApp
: Starting SpringBootApp on Jamess-MacBook-Pro.local with PID 11480
(.../target/springboot-app-example-6.1.0-SNAPSHOT.jar ...)
2019-09-06 15:56:45.668 INFO 11480 --- [           main]
i.e.s.examples.app.SpringBootApp
: No active profile set, falling back to default profiles: default
2019-09-06 15:56:46.146 INFO 11480 --- [           main]
i.e.s.examples.app.SpringBootApp
: Started SpringBootApp in 5.791 seconds (JVM running for 6.161) ③
Hello []                                ④ ⑤
Done SpringApplication                   ⑥
```

- ① Our `SpringBootApp.main()` is called and logs `Running SpringApplication`
 - ② `SpringApplication.run()` is called to execute the Spring Boot application
 - ③ Our `AppCommand` component is found within the classpath at or under the package declaring `@SpringBootApplication`
 - ④ The `AppCommand` component `run()` method is called, and it prints out a message
 - ⑤ The Spring Boot application terminates
 - ⑥ Our `SpringBootApp.main()` logs `Done SpringApplication` and exits

33.1. Implementation Note

I added print statements directly in the Spring Boot Application's main() method to help illustrate when calls were made. This output could have been packaged into listener callbacks to leave the `main()` method implementation free—except to register the callbacks. If you happen to need more complex behavior to fire before the Spring context begins initialization, then look to add `listeners` of the `SpringApplication` instead.



Chapter 34. Configure pom.xml to Test

At this point we are again ready to set up an automated execution of our JAR as a part of the build. We can do that by adding a separate goal execution of the `spring-boot-maven-plugin`.

```
<build>
  ...
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <executions>
        <execution>
          <id>run-application</id> ①
          <phase>integration-test</phase>
          <goals>
            <goal>run</goal>
          </goals>
          <configuration> ②
            <arguments>Maven,plugin-supplied,args</arguments>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

① new execution of the `run` goal to be performed during the Maven `integration-test` phase

② command line arguments passed to main

- Phase order

1. ...
2. `package`
3. `pre-integration`
4. `integration-test`
5. `post-integration`
6. `verify`
7. ...

34.1. Execute JAR as part of the build

```
$ mvn clean verify
[INFO] Scanning for projects...
...
[INFO] --- spring-boot-maven-plugin:3.3.2:run (run-application) @ springboot-app-
```

```
example ---  
[INFO] Attaching agents: [] ①  
Running SpringApplication
```

\\ / _____()_____\n(()__ | ' _ | ' | | ' _ \\\` | \\ \\ \\ \\ \\\n\\ \\ _ _ _) | () | | | | | | (| |)))))\n' | _ _ _ | . _ _ | _ _ | _ \\\n=====|_|=====|__/_=/__/_/_/\n:: Spring Boot :: (v3.3.2)

```
2022-07-02 14:11:46.110 INFO 48432 --- [           main]
i.e.e.a.build.springboot.SpringBootApp : Starting SpringBootApp using Java 17.0.3 on
Jamess-MacBook-Pro.local with PID 48432 (.../springboot-app-example/target/classes
started by jim in .../springboot-app-example)
2022-07-02 14:11:46.112 INFO 48432 --- [           main]
i.e.e.a.build.springboot.SpringBootApp : No active profile set, falling back to 1
default profile: "default"
2022-07-02 14:11:46.463 INFO 48432 --- [           main]
i.e.e.a.build.springboot.SpringBootApp : Started SpringBootApp in 0.611 seconds (JVM
running for 0.87)
Component code says Hello [Maven, plugin-supplied, args] ②
Done SpringApplication
```

① Our plugin is executing

② Our application was executed and the results displayed

Chapter 35. Summary

As a part of this material, the student has learned how to:

1. Add Spring Boot constructs and artifact dependencies to the Maven POM
2. Define Application class with a main() method
3. Annotate the application class with `@SpringBootApplication` (and optionally use lower-level annotations)
4. Place the application class in a Java package that is at or above the Java packages with beans that will make up the core of your application
5. Add component classes that are core to your application to your Maven module
6. Typically, define components in a Java package that is at or below the Java package for the `SpringBootApplication`
7. Annotate components with `@Component` (or other special-purpose annotations used by Spring)
8. Execute application like a normal executable JAR

Assignment 0: Application Build

copyright © 2024 jim stafford (jim.stafford@jhu.edu)

The following makes up "Assignment 0". It is intended to get you started developing right away, communicating questions/answers, and turning something in with some of the basics.

As with most assignments, a set of starter projects is available in [assignment-starter/autorentals-starter](#). It is expected that you can implement the complete assignment on your own. However, the Maven poms and the portions unrelated to the assignment focus are commonly provided for reference to keep the focus on each assignment part. Your submission should not be a direct edit/hand-in of the starters. Your submission should—at a minimum:

- be in a separate source tree—not within the class examples
- have a local parent pom.xml that extends either [spring-boot-starter-parent](#) or [ejava-build-parent](#)
- use your own Maven groupIds
 - change the "starter" portion of the provided groupId to a name unique to you

```
Change: <groupId>info.ejava-student.starter.assignments.projectName</groupId>
To:      <groupId>info.ejava-student.[your-
value].assignments.projectName</groupId>
```

- use your own Maven descriptive name
 - change the "Starter" portion of the provided name to a name unique to you

```
Change: <name>Starter::Assignments::ProjectName</name>
To:      <name>[Your Value]::Assignments::ProjectName</name>
```

- use your own Java package names
 - change the "starter" portion of the provided package name to a name unique to you

```
Change: package info.ejava_student.starter.assignment0.app.autorentals;
To:      package info.ejava_student.[your_value].assignment0.app.autorentals;
```

The following diagram depicts the 3 modules (parent, javaapp, and bootapp) you will turn in. You will inherit or depend on external artifacts that will be supplied via Maven.

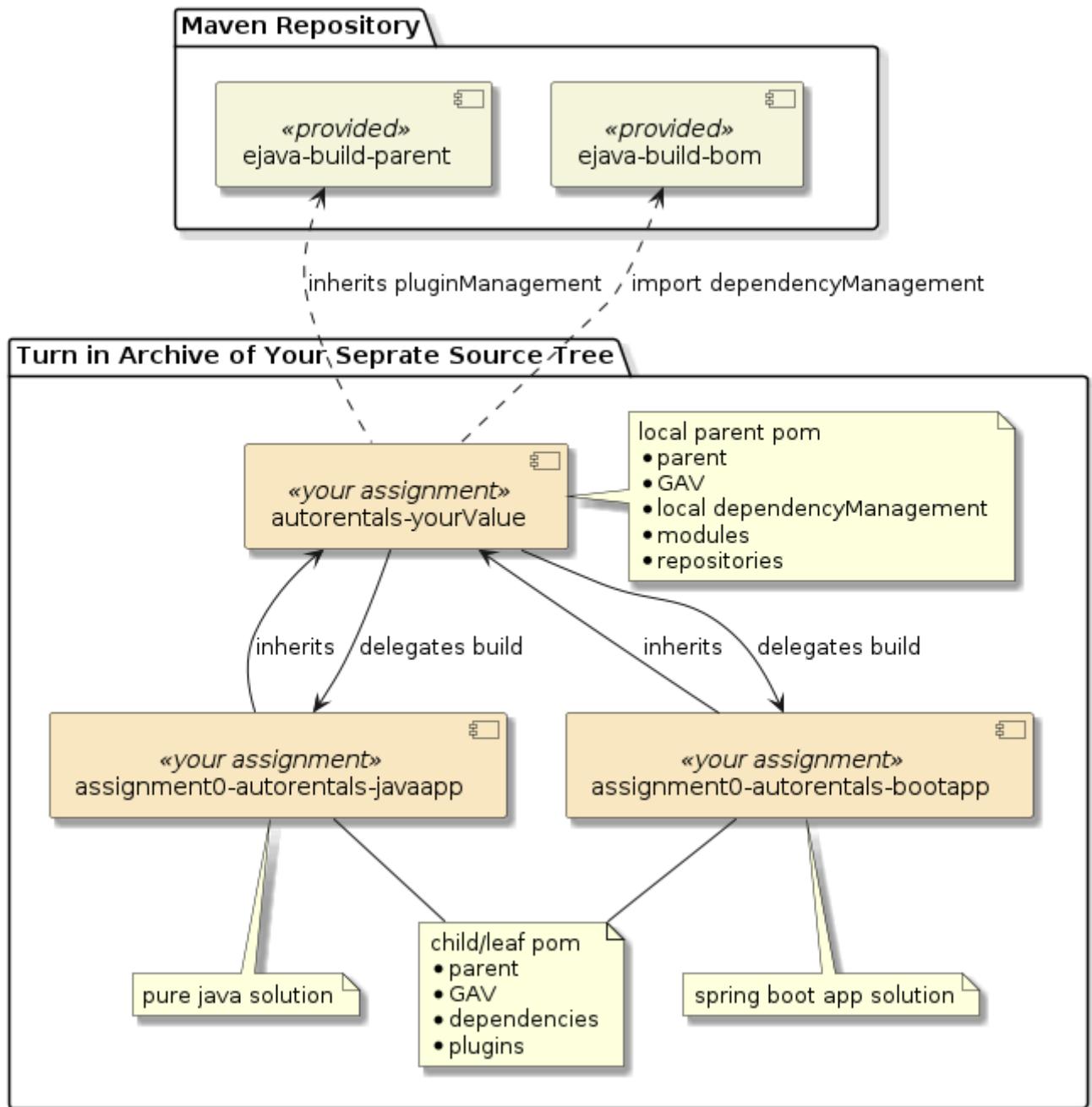


Figure 8. Assignment Tree

Chapter 36. Part A: Build Pure Java Application JAR

36.1. Purpose

In this portion of the assignment, you will demonstrate your knowledge of building a module containing a pure Java application. You will:

1. create source code for an executable Java class
2. add that Java class to a Maven module
3. build the module using a Maven pom.xml
4. execute the application using a classpath
5. configure the application as an executable JAR
6. execute an application packaged as an executable JAR

36.2. Overview

In this portion of the assignment you are going to implement a JAR with a Java main class and execute it.

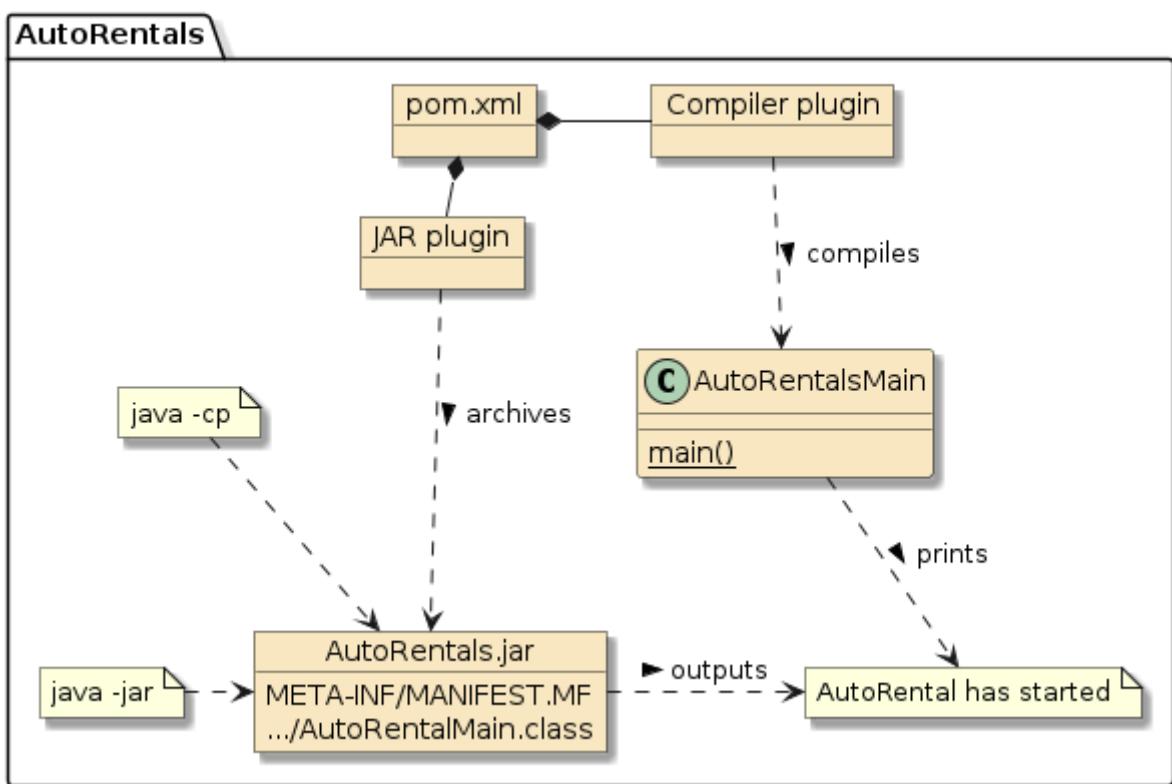


Figure 9. Pure Java Main Application

i In this part of the assignment, the name of the main class is shown specialized to "AutoRentals" since it contains (or directly calls) code having specifically to do with

"AutoRentals".

36.3. Requirements

1. Create a Maven project that will host a Java program
2. Supply a single Java class with a main() method that will print a single "AutoRental has started" message to stdout
3. Compile the Java class
4. Archive the Java class into a JAR
5. Execute the Java class using the JAR as a classpath
6. Register the Java class as the **Main-Class** in the **META-INF/MANIFEST.MF** file of the JAR
7. Execute the JAR to launch the Java class
8. Turn in a source tree with a complete Maven module that will build and execute a demonstration of the pure Java main application.

36.4. Grading

Your solution will be evaluated on:

1. create source code for an executable Java class
 - a. whether the Java class includes a non-root Java package
 - b. the assignment of a unique Java package for your work
 - c. whether you have successfully provided a main method that prints a startup message
2. add that Java class to a Maven module
 - a. the assignment of a unique groupId relative to your work
 - b. whether the module follows standard, basic Maven src/main directory structure
3. build the module using a Maven pom.xml
 - a. whether the module builds from the command line
4. execute the application using a classpath
 - a. if the Java main class executes using a **java -cp** approach
 - b. if the demonstration of execution is performed as part of the Maven build
5. execute an application packaged as an executable JAR
 - a. if the java main class executes using a **java -jar** approach
 - b. if the demonstration of execution is performed as part of the Maven build

36.5. Additional Details

1. The root maven pom can extend either **spring-boot-starter-parent** or **ejava-build-parent**. Add **<relativeParent/>** tag to parent reference to indicate an orphan project if doing so.

- When inheriting or depending on `ejava` class modules, include a JHU repository reference in your root pom.xml.

```
<repositories>
  <repository>
    <id>ejava-nexus-snapshots</id>
    <url>https://pika.jhuep.com/nexus/repository/ejava-snapshots</url>
  </repository>
</repositories>
```

- The maven build shall automate to demonstration of the two execution styles. You can use the `maven-antrun-plugin` or any other Maven plugin to implement this.
- A quick start project is available in `assignment-starter/autorentals-starter/assignment0-autorentals-javaapp` Modify Maven groupId and Java package if used.

Chapter 37. Part B: Build Spring Boot Executable JAR

37.1. Purpose

In this portion of the assignment you will demonstrate your knowledge of building a simple Spring Boot Application. You will:

1. construct a basic Spring Boot application
2. define a simple Spring component and inject that into the Spring Boot application
3. build and execute an executable Spring Boot JAR

37.2. Overview

In this portion of the assignment, you are going to implement a Spring Boot executable JAR with a Spring Boot application and execute it.

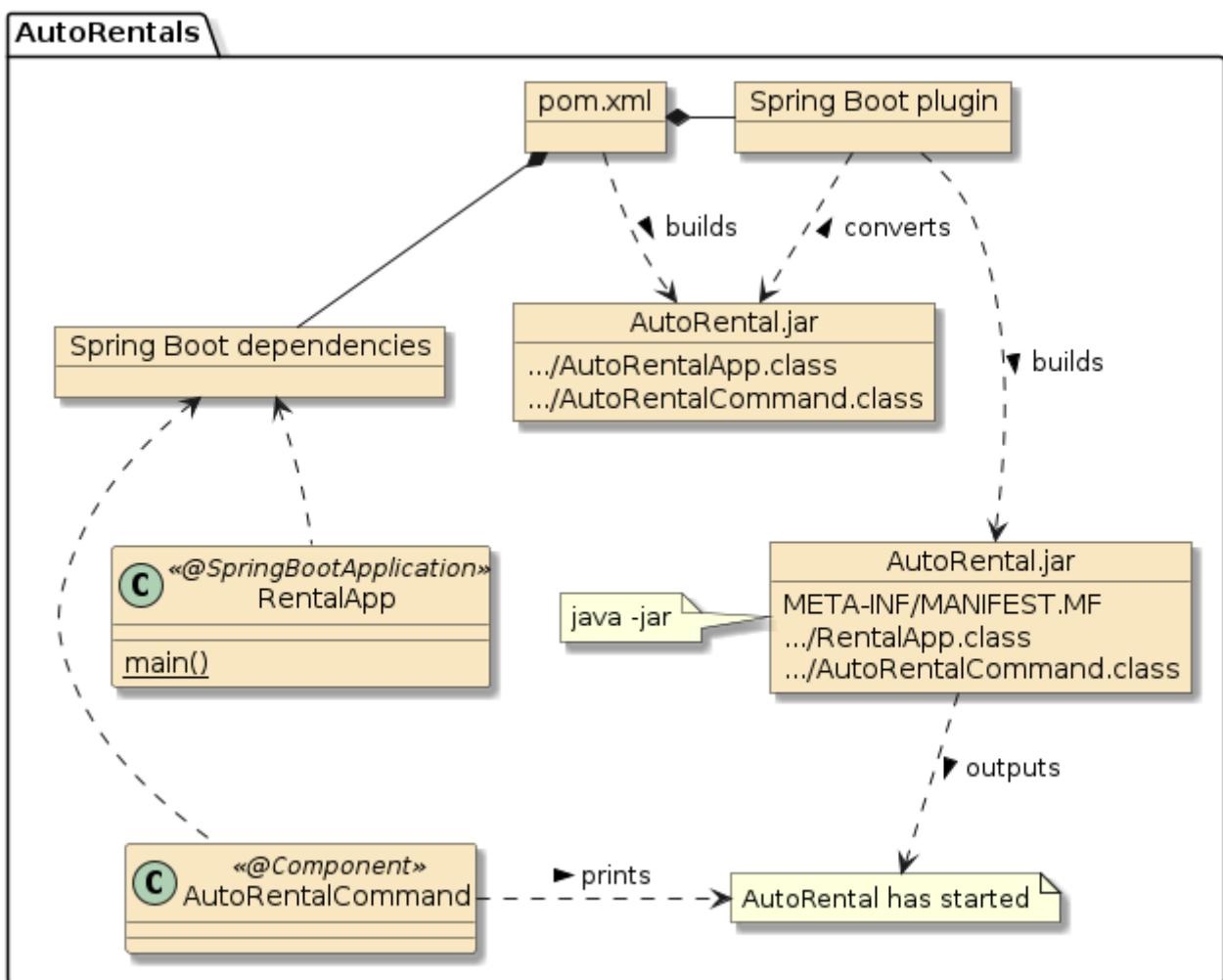


Figure 10. Spring Boot Application



In this part of the assignment, the name of the main class can be generalized to "RentalApp" since nothing about that class specifically has or calls anything to do

with "AutoRentals". All details of the specialization are within the "AutoRentalCommand" component.

37.3. Requirements

1. Create a Maven project to host a Spring Boot Application
2. Supply a single Java class with a main() method that bootstraps the Spring Boot Application
3. Supply a `@Component` that will be loaded and invoked when the application starts
 - a. have that `@Component` print a single "AutoRental has started" message to stdout
4. Compile the Java class
5. Archive the Java class
6. Convert the JAR into an executable Spring Boot Application JAR
7. Execute the JAR and Spring Boot Application
8. Turn in a source tree with a complete Maven module that will build and execute a demonstration of the Spring Boot application

37.4. Grading

Your solution will be evaluated on:

1. extend the standard Maven jar module packaging type to include core Spring Boot dependencies
 - a. whether you have added a dependency on `spring-boot-starter` (directly or indirectly) to bring in required dependencies
2. construct a basic Spring Boot application
 - a. whether you have defined a proper `@SpringBootApplication`
3. define a simple Spring component and inject that into the Spring Boot application
 - a. whether you have successfully injected a `@Component` that prints a startup message
4. build and execute an executable Spring Boot JAR
 - a. whether you have configured the Spring Boot plugin to build an executable JAR
 - b. if the demonstration of execution is performed as part of the Maven build

37.5. Additional Details

1. The root maven pom can extend either `spring-boot-starter-parent` or `ejava-build-parent`. Add `<relativeParent/>` tag to parent reference to indicate an orphan project if doing so.
2. When inheriting or depending on `ejava` class modules, include a JHU repository reference in your root pom.xml.

`<repositories>`

```
<repository>
  <id>ejava-nexus-snapshots</id>
  <url>https://pika.jhuep.com/nexus/repository/ejava-snapshots</url>
</repository>
</repositories>
```

3. The maven build shall automate to demonstration of the application using the **spring-boot-maven-plugin**. There is no need for the **maven-antrun-plugin** in this portion of the assignment.
4. A quick start project is available in **assignment-starter/autorentals-starter/assignment0-autorentals-bootapp**. Modify Maven groupId and Java package if used.

Bean Factory and Dependency Injection

copyright © 2024 jim stafford (jim.stafford@jhu.edu)

Chapter 38. Introduction

This material provides an introduction to configuring an application using a factory method. This is the most basic use of separation between the interface used by the application and the decision of what the implementation will be.

The configuration choice shown will be part of the application but as you will see later, configurations can be deeply nested—far away from the details known to the application writer.

38.1. Goals

The student will learn:

- to decouple an application through the separation of interface and implementation
- to configure an application using dependency injection and factory methods of a configuration class

38.2. Objectives

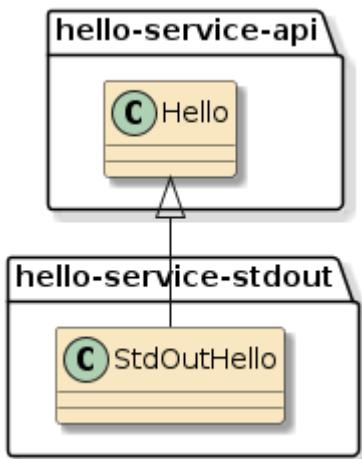
At the conclusion of this lecture and related exercises, the student will be able to:

1. implement a service interface and implementation component
2. package a service within a Maven module separate from the application module
3. implement a Maven module dependency to make the component class available to the application module
4. use a `@Bean` factory method of a `@Configuration` class to instantiate a Spring-managed component

Chapter 39. Hello Service

To get started, we are going to create a simple `Hello` service. We are going to implement an interface and a single implementation right off the bat. They will be housed in two separate modules:

- `hello-service-api`
- `hello-service-stdout`



We will start out by creating two separate module directories.

39.1. Hello Service API

The Hello Service API module will contain a single interface and `pom.xml`.

```
hello-service-api/
|-- pom.xml
`-- src
  '-- main
    '-- java
      '-- info
        '-- ejava
          '-- examples
            '-- app
              '-- hello
                '-- Hello.java ①
```

① Service interface

39.2. Hello Service StdOut

The Hello Service StdOut module will contain a single implementation class and `pom.xml`.

```
hello-service-stdout/
|-- pom.xml
`-- src
```

```
'-- main
  '-- java
    '-- info
      '-- ejava
        '-- examples
          '-- app
            '-- hello
              '-- stdout
                '-- StdOutHello.java ①
```

① Service implementation

39.3. Hello Service API pom.xml

We will be building a normal Java JAR with no direct dependencies on Spring Boot or Spring.

hello-service-api pom.xml

```
#pom.xml
...
<groupId>info.ejava.examples.app</groupId>
<version>6.1.0-SNAPSHOT</version>
<artifactId>hello-service-api</artifactId>
<packaging>jar</packaging>
...
```

39.4. Hello Service StdOut pom.xml

The implementation will be similar to the interface's pom.xml except it requires a dependency on the interface module.

hello-service-stdout pom.xml

```
#pom.xml
...
<groupId>info.ejava.examples.app</groupId>
<version>6.1.0-SNAPSHOT</version>
<artifactId>hello-service-stdout</artifactId>
<packaging>jar</packaging>

<dependencies>
  <dependency>
    <groupId>${project.groupId}</groupId> ①
    <artifactId>hello-service-api</artifactId>
    <version>${project.version}</version> ①
  </dependency>
</dependencies>
...
```

① Dependency references leveraging \${project} variables module shares with dependency



Since we are using the same source tree, we can leverage \${project} variables. This will not be the case when declaring dependencies on external modules.

39.5. Hello Service Interface

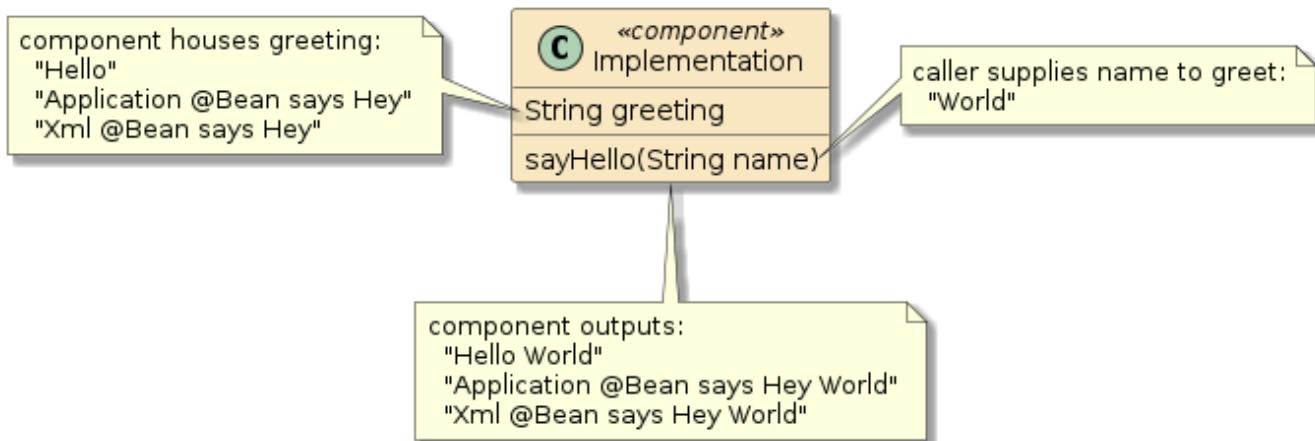
The interface is quite simple, just pass in the String name for what you want the service to say hello to.

```
package info.ejava.examples.app.hello;

public interface Hello {
    void sayHello(String name);
}
```

The service instance will be responsible for

- the greeting
- the implementation — how we say hello



39.6. Hello Service Sample Implementation

Our sample implementation is just as simple. It maintains the greeting in a final instance attribute and uses `stdout` to print the message.

```
package info.ejava.examples.app.hello.stdout; ①

public class StdOutHello implements Hello {
    private final String greeting; ②

    public StdOutHello(String greeting) { ③
        this.greeting = greeting;
    }
}
```

```

@Override ④
public void sayHello(String name) {
    System.out.println(greeting + " " + name);
}

```

- ① Implementation defined within own package
- ② `greeting` will hold our phrase for saying hello and is made final to highlight it is required and will not change during the lifetime of the class instance
- ③ A single constructor is provided to define a means to initialize the instance. Remember—the `greeting` is final and must be set during class instantiation and not later during a setter.
- ④ The `sayHello()` method provides implementation of method defined in interface



`final` requires the value set when the instance is created and never change



Spring recommends constructor injection. This provides an immutable object and better assures that required dependencies are not null. ^[1]

39.7. Hello Service Modules Complete

We are now done implementing our sample service interface and implementation. We just need to build and install it into the repository to make available to the application.

39.8. Hello Service API Maven Build

```

$ mvn clean install -f hello-service-api
[INFO] Scanning for projects...
[INFO]
[INFO] -----< info.ejava.examples.app:hello-service-api >-----
[INFO] Building App::Config::Hello Service API 6.1.0-SNAPSHOT
[INFO] -----[ jar ]-----
[INFO]
[INFO] --- maven-clean-plugin:3.1.0:clean (default-clean) @ hello-service-api ---
[INFO]
[INFO] --- maven-resources-plugin:3.1.0:resources (default-resources) @ hello-service-
api ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory .../app-config/hello-service-
api/src/main/resources
[INFO]
[INFO] --- maven-compiler-plugin:3.8.1:compile (default-compile) @ hello-service-api
---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 1 source file to .../app-config/hello-service-api/target/classes
[INFO]
[INFO] --- maven-resources-plugin:3.1.0:testResources (default-testResources) @ hello-

```

```
service-api ---  
[INFO] Using 'UTF-8' encoding to copy filtered resources.  
[INFO] skip non existing resourceDirectory .../app-config/hello-service-  
api/src/test/resources  
[INFO]  
[INFO] --- maven-compiler-plugin:3.8.1:testCompile (default-testCompile) @ hello-  
service-api ---  
[INFO] No sources to compile  
[INFO]  
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ hello-service-api ---  
[INFO] No tests to run.  
[INFO]  
[INFO] --- maven-jar-plugin:3.1.2:jar (default-jar) @ hello-service-api ---  
[INFO] Building jar: .../app-config/hello-service-api/target/hello-service-api-6.1.0-  
SNAPSHOT.jar  
[INFO]  
[INFO] --- maven-install-plugin:3.0.0-M1:install (default-install) @ hello-service-api  
---  
[INFO] Installing .../app-config/hello-service-api/target/hello-service-api-6.1.0-  
SNAPSHOT.jar to ....m2/repository/info/ejava/examples/app/hello-service-api/6.1.0-  
SNAPSHOT/hello-service-api-6.1.0-SNAPSHOT.jar  
[INFO] Installing .../app-config/hello-service-api/pom.xml to  
....m2/repository/info/ejava/examples/app/hello-service-api/6.1.0-SNAPSHOT/hello-  
service-api-6.1.0-SNAPSHOT.pom  
[INFO] -----  
[INFO] BUILD SUCCESS  
[INFO] -----  
[INFO] Total time: 2.070 s
```

39.9. Hello Service StdOut Maven Build

```
$ mvn clean install -f hello-service-stdout  
[INFO] Scanning for projects...  
[INFO]  
[INFO] -----< info.ejava.examples.app:hello-service-stdout >-----  
[INFO] Building App::Config::Hello Service StdOut 6.1.0-SNAPSHOT  
[INFO] -----[ jar ]-----  
[INFO]  
[INFO] --- maven-clean-plugin:3.1.0:clean (default-clean) @ hello-service-stdout ---  
[INFO]  
[INFO] --- maven-resources-plugin:3.1.0:resources (default-resources) @ hello-service-  
stdout ---  
[INFO] Using 'UTF-8' encoding to copy filtered resources.  
[INFO] skip non existing resourceDirectory .../app-config/hello-service-  
stdout/src/main/resources  
[INFO]  
[INFO] --- maven-compiler-plugin:3.8.1:compile (default-compile) @ hello-service-  
stdout ---  
[INFO] Changes detected - recompiling the module!
```

```
[INFO] Compiling 1 source file to .../app-config/hello-service-stdout/target/classes
[INFO]
[INFO] --- maven-resources-plugin:3.1.0:testResources (default-testResources) @ hello-
service-stdout ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory .../app-config/hello-service-
stdout/src/test/resources
[INFO]
[INFO] --- maven-compiler-plugin:3.8.1:testCompile (default-testCompile) @ hello-
service-stdout ---
[INFO] No sources to compile
[INFO]
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ hello-service-stdout ---
[INFO] No tests to run.
[INFO]
[INFO] --- maven-jar-plugin:3.1.2:jar (default-jar) @ hello-service-stdout ---
[INFO] Building jar: .../app-config/hello-service-stdout/target/hello-service-stdout-
6.1.0-SNAPSHOT.jar
[INFO]
[INFO] --- maven-install-plugin:3.0.0-M1:install (default-install) @ hello-service-
stdout ---
[INFO] Installing .../app-config/hello-service-stdout/target/hello-service-stdout-
6.1.0-SNAPSHOT.jar to .../.m2/repository/info/ejava/examples/app/hello-service-
stdout/6.1.0-SNAPSHOT/hello-service-stdout-6.1.0-SNAPSHOT.jar
[INFO] Installing .../app-config/hello-service-stdout/pom.xml to
.../.m2/repository/info/ejava/examples/app/hello-service-stdout/6.1.0-SNAPSHOT/hello-
service-stdout-6.1.0-SNAPSHOT.pom
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 2.658 s
```

[1] https://docs.spring.io/spring-framework/reference/core/beans/dependencies/factory-collaborators.html_Constructor-based or
setter-based DI?_, Spring.io

Chapter 40. Application Module

We now move on to developing our application within its own module containing two (2) classes similar to earlier examples.

```
|-- pom.xml
`-- src
  '-- main
    '-- java
      '-- info
        '-- ejava
          '-- examples
            '-- app
              '-- config
                '-- beanfactory
                  |-- AppCommand.java ②
                  '-- SelfConfiguredApp.java ①
```

① Class with Java main() that starts Spring

② Class containing our first component that will be the focus of our injection

40.1. Application Maven Dependency

We make the Hello Service visible to our application by adding a dependency on the `hello-service-api` and `hello-service-stdout` artifacts. Since the implementation already declares a compile dependency on the interface, we can get away with only declaring a direct dependency just on the implementation.

```
<groupId>info.ejava.examples.app</groupId>
<artifactId>appconfig-beanfactory-example</artifactId>
<name>App::Config::Bean Factory Example</name>

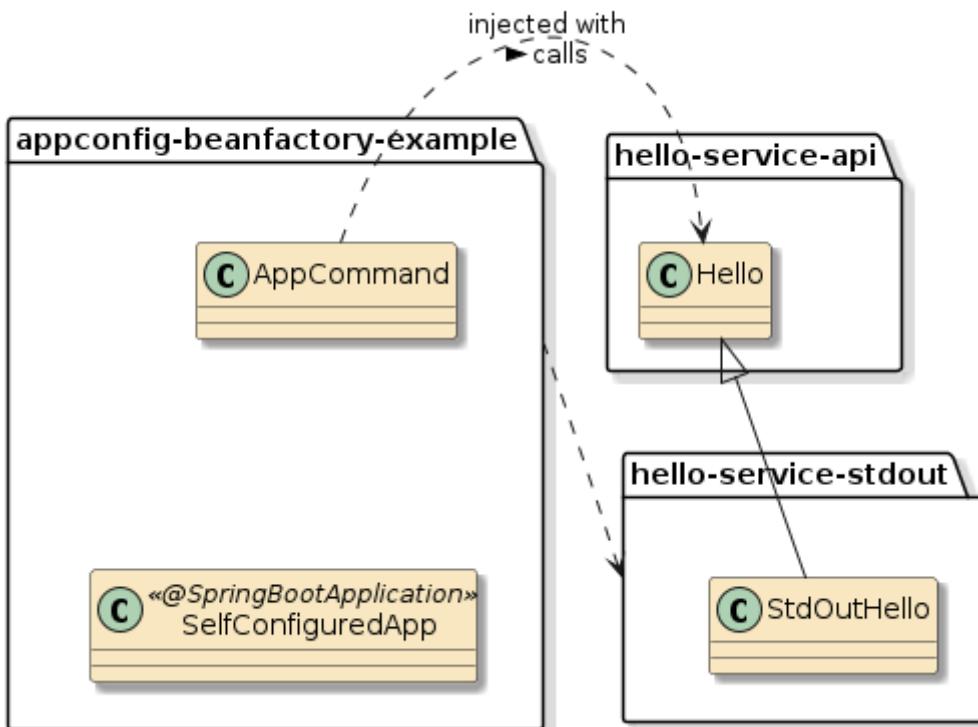
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
  </dependency>
  <dependency>
    <groupId>${project.groupId}</groupId>
    <artifactId>hello-service-stdout</artifactId> ①
    <version>${project.version}</version>
  </dependency>
</dependencies>
```

① Dependency on implementation creates dependency on both implementation and interface



In this case, the module we are depending upon is in the same `groupId` and shares

the same **version**. For simplicity of reference and versioning, I used the `${project}` variables to reference it. That will not always be the case.



40.2. Viewing Dependencies

You can verify the dependencies exist using the `tree` goal of the `dependency` plugin.

Artifact Dependency Tree

```
$ mvn dependency:tree -f hello-service-stdout
...
[INFO] --- maven-dependency-plugin:3.1.1:tree (default-cli) @ hello-service-stdout ---
[INFO] info.ejava.examples.app:hello-service-stdout:jar:6.1.0-SNAPSHOT
[INFO] \- info.ejava.examples.app:hello-service-api:jar:6.1.0-SNAPSHOT:compile
```

40.3. Application Java Dependency

Next we add a reference to the `Hello` interface and define how we can get it injected. In this case we are using constructor injection where the instance is supplied to the class through a parameter to the constructor.

i The component class now has a non-default constructor to allow the `Hello` implementation to be injected and the Java attribute is defined as `final` to help assure that the value is assigned during the constructor.

```
package info.ejava.examples.app.config.beanfactory;

import org.springframework.boot.CommandLineRunner;
```

```
import org.springframework.stereotype.Component;  
  
import info.ejava.examples.app.hello.Hello;  
  
@Component  
public class AppCommand implements CommandLineRunner {  
    private final Hello greeter; ①  
  
    public AppCommand(Hello greeter) { ②  
        this.greeter = greeter;  
    }  
  
    public void run(String... args) throws Exception {  
        greeter.sayHello("World");  
    }  
}
```

- ① Add a reference to the Hello interface. Java attribute defined as `final` to help assure that the value is assigned during the constructor.
- ② Using constructor injection where the instance is supplied to the class through a parameter to the constructor

Chapter 41. Dependency Injection

Our `AppCommand` class has been defined only with the interface to `Hello` and not a specific implementation.

This Separation of Concerns helps improve modularity, testability, reuse, and many other desirable features of an application. The interaction between the two classes is defined by an interface.

But how do does our client class (`AppCommand`) get an instance of the implementation (`StdOutHello`)?

- If the client class directly instantiates the implementation—it is coupled to that specific implementation.

```
public AppCommand() {  
    this greeter = new StdOutHello("World");  
}
```

- If the client class procedurally delegates to a factory—it runs the risk of violating Separation of Concerns by adding complex initialization code to its primary business purpose

```
public AppCommand() {  
    this greeter = BeanFactory.makeGreeter();  
}
```

Traditional procedural code normally makes calls to libraries in order to perform a specific purpose. If we instead remove the instantiation logic and decisions from the client and place that elsewhere, we can keep the client more focused on its intended purpose. With this inversion of control (IoC), the application code is part of a framework that calls the application code when it is time to do something versus the other way around. In this case the framework is for application assembly.

Most frameworks, including Spring, implement dependency injection through a form of IoC.

Chapter 42. Spring Dependency Injection

We defined the dependency using the `Hello` interface and have three primary ways to have dependencies injected into an instance.

```
import org.springframework.beans.factory.annotation.Autowired;

public class AppCommand implements CommandLineRunner {
    // @Autowired -- FIELD injection ③
    private Hello greeter;

    @Autowired // -- Constructor injection ①
    public AppCommand(Hello greeter) {
        this.greeter = greeter;
    }

    // @Autowired -- PROPERTY injection ②
    public void setGreeter(Hello hello) {
        this.greeter = hello;
    }
}
```

① constructor injection - injected values required prior to instance being created

② field injection - value injected directly into attribute

③ setter or property injection - `setter()` called with value

42.1. @Autowired Annotation

The `@Autowired(required=…)` annotation

- may be applied to fields, methods, constructors
- `@Autowired(required=true)` - default value for `required` attribute
 - successful injection mandatory when applied to a property
 - specific constructor use required when applied to a constructor
 - only a single constructor per class may have this annotation
- `@Autowired(required=false)`
 - injected bean not required to exist when applied to a property
 - specific constructor an option for container to use
 - multiple constructors may have this annotation applied
 - container will determine best based on number of matches
 - **single constructor has an implied `@Autowired(required=false)`** - making annotation optional

There are more details to learn about injection and the lifecycle of a bean. However, know that we

are using constructor injection at this point in time since the dependency is required for the instance to be valid.

42.2. Dependency Injection Flow

In our example:

- Spring will detect the AppCommand component and look for ways to instantiate it
- The only constructor requires a Hello instance
- Spring will then look for a way to instantiate an instance of Hello

Chapter 43. Bean Missing

When we go to run the application, we get the following error

```
$ mvn clean package  
...  
*****  
APPLICATION FAILED TO START  
*****
```

Description:

Parameter 0 of constructor in AppCommand required a bean of type 'Hello' that could not be found.

Action:

Consider defining a bean of type 'Hello' in your configuration.

The problem is that the container has no knowledge of any beans that can satisfy the only available constructor. The `StdOutHello` class is not defined in a way that allows Spring to use it.

43.1. Bean Missing Error Solution(s)

We can solve this in at least two (2) ways.

1. Add `@Component` to the `StdOutHello` class. This will trigger Spring to directly instantiate the class.

```
@Component  
public class StdOutHello implements Hello {
```

- problem: It may be one of many implementations of Hello

2. Define what is needed using a `@Bean` factory method of a `@Configuration` class. This will trigger Spring to call a method that is in charge of instantiating an object of the type identified in the method return signature.

```
@Configuration  
public class AConfigurationClass {  
    @Bean  
    public Hello hello() {  
        return new StdOutHello("...");  
    }  
}
```

Chapter 44. @Configuration classes

@Configuration classes are classes that Spring expects to have one or more @Bean factory methods. If you remember back, our Spring Boot application class was annotated with @SpringBootApplication

```
@SpringBootApplication ①
//==> wraps @SpringBootConfiguration ②
//  ==> wraps @Configuration
public class SelfConfiguredApp {
    public static void main(String...args) {
        SpringApplication.run(SelfConfiguredApp.class, args);
    }
    //...
}
```

① @SpringBootApplication is a wrapper around a few annotations including @SpringBootConfiguration

② @SpringBootConfiguration is an alternative annotation to using @Configuration with the caveat that there be only one @SpringBootConfiguration per application

Therefore, we have the option to use our Spring Boot application class to host the configuration and the @Bean factory.

Chapter 45. @Bean Factory Method

There is more to `@Bean` factory methods than we will cover here, but at its simplest and most functional level—this is a series of factory methods the container will call to instantiate components for the application. By default they are all eagerly instantiated and the dependencies between them are resolved (if resolvable) by the container.

Adding a `@Bean` factory method to our Spring Boot application class will result in the following in our Java class.

```
@SpringBootApplication ④ ⑤
public class SelfConfiguredApp {
    public static void main(String...args) {
        SpringApplication.run(SelfConfiguredApp.class, args);
    }

    @Bean ①
    public Hello hello() { ②
        return new StdOutHello("Application @Bean says Hey"); ③
    }
}
```

- ① method annotated with `@Bean` implementation
- ② method returns `Hello` type required by container
- ③ method returns a fully instantiated instance.
- ④ method hosted within class with `@Configuration` annotation
- ⑤ `@SpringBootConfiguration` annotation included the capability defined for `@Configuration`



Anything missing to create instance gets declared as an input to the method and, it will get created in the same manner and passed as a parameter.

Chapter 46. @Bean Factory Used

With the `@Bean` factory method in place, all comes together at runtime to produce the following:

```
$ java -jar target/appconfig-beanfactory-example-*-SNAPSHOT-bootexec.jar  
...  
Application @Bean says Hey World
```

- the container
 - obtained an instance of a `Hello` bean
 - passed that bean to the `AppCommand` class' constructor to instantiate that `@Component`
- the `@Bean` factory method
 - chose the implementation of the `Hello` service (`StdOutHello`)
 - chose the greeting to be used ("Application @Bean says Hey")

```
return new StdOutHello("Application @Bean says Hey");
```

- the `AppCommand` `CommandLineRunner` determined who to say hello to ("World")

```
greeter.sayHello("World");
```

Chapter 47. Factory Alternative: XML Configuration

Although most developments today prefer Java-based configurations, the legacy approach of defining beans using XML is still available.

To do so, we define an `@ImportResource` annotation on a `@Configuration` class that references pathnames using either a class or file path. In this example we are referencing a file called `applicationContext.xml` in the `resources` package within the classpath.

```
import org.springframework.context.annotation.ImportResource;

@SpringBootApplication
@ImportResource({"classpath:contexts/applicationContext.xml"}) ①
public class XmlConfiguredApp {
    public static void main(String...args) {
        SpringApplication.run(XmlConfiguredApp.class, args);
    }
}
```

① `@ImportResource` will enact the contents of `context/applicationContext.xml`

The XML file can be placed inside the JAR of the application module by adding it to the `src/main/resources` directory of this or other modules in our classpath.

```
|-- pom.xml
\-- src
  '-- main
    '-- java
      '-- info
        '-- ejava
          '-- examples
            '-- app
              '-- config
                '-- xmlconfig
                  |-- AppCommand.java
                  '-- XmlConfiguredApp.java
    '-- resources
      '-- contexts
        '-- applicationContext.xml
```

```
$ jar tf target/appconfig-xmlconfig-example-* -SNAPSHOT-bootexec.jar | grep
applicationContext.xml
BOOT-INF/classes/contextes/applicationContext.xml
```

The XML file has a specific `schema` to follow. It can be every bit as powerful as Java-based configurations and have the added feature that it can be edited without recompilation of a Java

class.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean class="info.ejava.examples.app.hello.stdout.StdOutHello"> ①
        <constructor-arg value="Xml @Bean says Hey" /> ②
    </bean>
</beans>
```

- ① A specific implementation of the `Hello` interface is defined
- ② Text is injected into the constructor when container instantiates

This produces the same relative result as the Java-based configuration.

```
$ java -jar target/appconfig-xmlconfig-example-*-SNAPSHOT-bootexec.jar
...
Xml @Bean says Hey World
```

Chapter 48. @Configuration Alternatives

With the basics of `@Configuration` classes understood, I want to introduce two other common options for a `@Bean` factory:

- scope
- proxyBeanMethods

By default,

- Spring will instantiate a single component to represent the bean (singleton)
- Spring will create a (CGLIB) proxy for each `@Configuration` class to assure the result is processed by Spring and that each bean client for singleton-scoped beans get the same copy. This proxy can add needless complexity depending on how sibling methods are designed.

The following concepts and issues will be discussed:

- shared (singleton) or unique (prototype) component instances
- (unnecessary) role of the Spring proxy (`proxyBeanMethods`)
- potential consequences of calling sibling `@Bean` methods over injection

48.1. Example Lifecycle POJO

To demonstrate, I created an example POJO that identifies its instance and tracks its component lifecycle.

- the Java constructor is called for each POJO instance created
- `@PostConstruct` methods are called by Spring after dependencies have been injected. If and when you see debug from the `init()`, you know the POJO has been made into a component, with the potential for Spring container interpose.

Example POJO Class Created and Used by @Bean Factory

```
public class Example {  
    private final int exampleValue;  
    public Example(int value) { this.exampleValue = value; } ①  
    @PostConstruct ②  
    void init() {  
        System.out.println("@PostConstruct called for: " + exampleValue);  
    }  
    public String toString() { return Integer.toString(exampleValue); }  
}
```

① Constructor will be called for every instance created

② `@PostConstruct` will only get called by when POJO becomes a component

48.2. Example Lifecycle @Configuration Class

To further demonstrate, I have added a `@Configuration` class with some options that will be changed during the example.

Example @Configuration and Source @Bean Factory

```
@Configuration //default proxyBeanMethods=true
//@Configuration(proxyBeanMethods = true)
//@Configuration(proxyBeanMethods = false)
public class BeanFactoryProxyBeansConfiguration {
    private int value = 0;

    @Bean //default is singleton
    //@Scope(ConfigurableBeanFactory.SCOPE_SINGLETON) //"singleton"
    Example bean() {
        return new Example(value++);
    }
}
```

48.3. Consuming @Bean Factories within Same Class

There are two pairs of `Example`-consuming `@Bean` factories: calling and injected.

- Calling `@Bean` factories make a direct call to the supporting `@Bean` factory method.
- Injected `@Bean` factories simply declare their requirement in method input parameters.

Example @Bean Consumers

```
@Bean ①
String calling1() { return "calling1=" + bean(); }

@Bean ①
String calling2() { return "calling2=" + bean(); }

@Bean ②
String injected1(Example bean) { return "injected1=" + bean; }

@Bean ②
String injected2(Example bean) { return "injected2=" + bean; }
```

① calling consumers call the sibling `@Bean` factory method directly

② injected consumers are passed an instance of requirements when called

48.4. Using Defaults (Singleton, Proxy)

By default:

- `@Bean` factories use singleton scope
- `@Configuration` classes use `proxyBeanMethods=true`

That means that:

- @Bean factory method will be called only once by Spring
- @Configuration class instance will be proxied and direct calls (`calling1` and `calling2`) will receive the same singleton result

Proxied @Configuration Class Emits Same Singleton to all Consumers

```
@PostConstruct called for: 0 ②  
calling1=0 ①  
calling2=0 ①  
injected1=0 ①  
injected2=0 ①
```

① only one POJO instance was created

② only one component was initialized

48.5. Prototype Scope, Proxy True

If we change component scope created by the `bean()` method to "prototype"...

@Bean factory Scope set to non-default "prototype"

```
@Bean  
@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE) //"prototype"  
Example bean() {  
    return new Example(value++);  
}
```

...we get a unique POJO instance/component for each consumer (calling and injected) while `proxyBeanMethods` is still `true`.

Different Instance Produced for each Consumer and all Consumer Types

```
@PostConstruct called for: 0 ②  
@PostConstruct called for: 1 ②  
@PostConstruct called for: 2 ②  
@PostConstruct called for: 3 ②  
calling1=0 ①  
calling2=1 ①  
injected1=2 ①  
injected2=3 ①
```

① unique POJO instances were created

② each instance was a component

48.6. Prototype Scope, Proxy False

If we drop the CGLIB proxy, our configuration instance gets lighter, but ...

Spring Proxy Off, Prototype Scope

```
@Configuration(proxyBeanMethods = false)
public class BeanFactoryProxyBeansConfiguration {
    @Bean @Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE) //"prototype"
    Example bean() {
```

...only injected consumers are given "component" beans. The "calling" consumers are given "POJO" beans, that lack the potential for interpose.

```
@PostConstruct called for: 2 ②
@PostConstruct called for: 3 ②
calling1=0 ①
calling2=1 ①
injected1=2 ①
injected2=3 ①
```

① each consumer is given a unique instance

② only the injected callers are given components (with interpose potential)

48.7. Singleton Scope, Proxy False

Keeping the proxy eliminated and reverting back to the default singleton scope for the bean ...

Spring Proxy Off, Singleton Scope

```
@Configuration(proxyBeanMethods = false)
public class BeanFactoryProxyBeansConfiguration {

    @Bean @Scope(ConfigurableBeanFactory.SCOPE_SINGLETON) //"singleton" - default
    Example bean() {
```

...shows that only the injected consumers are receiving a singleton instance—initialized as a component, with the potential for interpose.

```
@PostConstruct called for: 0 ③
calling1=1 ②
calling2=2 ②
injected1=0 ①
injected2=0 ①
```

① injected consumers get the same instance

- ② calling consumers get unique instances independent of `@Scope`
- ③ only the injected consumers are getting a component (with interpose potential)

48.8. @Configuration Takeaways

- Spring instantiates all components, by default, as singletons—with the option to instantiate unique instances on demand when `@Scope` is set to "prototype".
- Spring, by default, constructs a CGLIB proxy to enforce those semantics for both calling and injected consumers.
 - Since `@Configuration` classes are only called once at start-up, it can be a waste of resources to construct a CGLIB proxy.
 - Using injection-only consumers, with no direct calls to `@Configuration` class methods, eliminates the need for the proxy.
 - adding `proxyFactoryBeans=false` eliminates the CGLIB proxy. Spring will enforce semantics for injected consumers

Chapter 49. Summary

In this module we

- decoupled part of our application into three Maven modules (app, iface, and impl1)
- decoupled the implementation details (`StdOutHello`) of a service from the caller (`AppCommand`) of that service
- injected the implementation of the service into a component using constructor injection
- defined a `@Bean` factory method to make the determination of what to inject
- showed an alternative using XML-based configuration and `@ImportResource`
- explored the differences between calling and injected sibling component consumers

In future modules we will look at more detailed aspects of Bean lifecycle and `@Bean` factory methods. Right now we are focused on following a path to explore decoupling our the application even further.

Value Injection

copyright © 2024 jim stafford (jim.stafford@jhu.edu)

Chapter 50. Introduction

One of the things you may have noticed was the hard-coded string in the AppCommand class in the previous example.

```
public void run(String... args) throws Exception {  
    greeter.sayHello("World");  
}
```

Let's say we don't want the value hard-coded or passed in as a command-line argument. Let's go down a path that uses standard Spring value injection to inject a value from a property file.

Ref: [Spring Boot application.properties file by Daniel Olszewski](#)

50.1. Goals

The student will learn:

- how to configure an application using properties
- how to use different forms of injection

50.2. Objectives

At the conclusion of this lecture and related exercises, the student will be able to:

1. implement value injection into a Spring Bean attribute using
 - field injection
 - constructor injection
2. inject a specific value at runtime using a command line parameter
3. define a default value for the attribute
4. define property values for attributes of different type

Chapter 51. @Value Annotation

To inject a value from a property source, we can add the Spring `@Value` annotation to the component property.

```
package info.ejava.examples.app.config.valueinject;

import org.springframework.beans.factory.annotation.Value;
...
@Component
public class AppCommand implements CommandLineRunner {
    private final Hello greeter;

    @Value("${app.audience}") ②
    private String audience; ①

    public AppCommand(Hello greeter) {
        this.greeter = greeter;
    }

    public void run(String... args) throws Exception {
        greeter.sayHello(audience);
    }
}
```

① defining target of value as a FIELD

② using FIELD injection to directly inject into the field

There are no specific requirements for property names but there are some common conventions followed using `(prefix).(property)` to scope the property within a context.

- `app.audience`
- `logging.file.name`
- `spring.application.name`

51.1. Value Not Found

However, if the property is not defined anywhere the following ugly error will appear.

```
org.springframework.beans.factory.BeanCreationException: Error creating bean with name
'appCommand' defined in file [.../app/app-config/appconfig-valueinject-
example/target/classes/info/ejava/examples/app/config/valueinject/AppCommand.class]:
Unexpected exception during bean creation
...
Caused by: java.lang.IllegalArgumentException: Could not resolve placeholder
'app.audience' in value "${app.audience}"
```

51.2. Value Property Provided by Command Line

We can try to fix the problem by defining the property value on the command line

```
$ java -jar target/appconfig-valueinject-example-*-SNAPSHOT-bootexec.jar \
--app.audience="Command line World" ①
...
Application @Bean says Hey Command line World
```

① use double dash (--) and property name to supply property value

51.3. Default Value

We can defend against the value not being provided by assigning a default value where we declared the injection

```
@Value("${app.audience:Default World}") ①
private String audience;
```

① use :value to express a default value for injection

That results in the following output

Property Default

```
$ java -jar target/appconfig-valueinject-example-*-SNAPSHOT-bootexec.jar
...
Application @Bean says Hey Default World
```

Property Defined

```
$ java -jar target/appconfig-valueinject-example-*-SNAPSHOT-bootexec.jar \
--app.audience="Command line World"
...
Application @Bean says Hey Command line World
```

Chapter 52. Constructor Injection

In the above version of the example, we injected the `Hello` bean through the constructor and the `audience` property using FIELD injection. This means

- the value for `audience` attribute will not be known during the constructor
- the value for `audience` attribute cannot be made final

```
@Value("${app.audience}")
private String audience;

public AppCommand(Hello greeter) {
    this.greeter = greeter;
    greeter.sayHello(audience); //X-no ①
}
```

① `audience` value will be null when used in the constructor — when using FIELD injection

52.1. Constructor Injection Solution

An alternative to using `field` injection is to change it to `constructor` injection. This has the benefit of having all properties injected in time to have them declared final.

```
@Component
public class AppCommand implements CommandLineRunner {
    private final Hello greeter;
    private final String audience; ②
    public AppCommand(Hello greeter,
                      @Value("${app.audience:Default World}") String audience) {
        this.greeter = greeter;
        this.audience = audience; ①
    }
}
```

① `audience` value will be known when used in the constructor

② `audience` value can be optionally made final

Chapter 53. @PostConstruct

If field-injection is our choice, we can account for the late-arriving injections by leveraging `@PostConstruct`. The Spring container will call a method annotated with `@PostConstruct` after instantiation (ctor called) and properties fully injected.

```
import jakarta.annotation.PostConstruct;  
...  
@Component  
public class AppCommand implements CommandLineRunner {  
    private final Hello greeter; ①  
    @Value("${app.audience}")  
    private String audience; ②  
  
    @PostConstruct  
    void init() { ③  
        greeter.sayHello(audience); //yes-greeter and audience initialized  
    }  
    public AppCommand(Hello greeter) {  
        this.greeter = greeter;  
    }  
}
```

① constructor injection occurs first and in-time to declare attribute as `final`

② field and property-injection occurs next and can involve many properties

③ Container calls `@PostConstruct` when all injection complete

Chapter 54. Property Types

54.1. non-String Property Types

Properties can also express non-String types as the following example shows.

```
@Component
public class PropertyExample implements CommandLineRunner {
    private final String strVal;
    private final int intValue;
    private final boolean booleanVal;
    private final float floatVal;

    public PropertyExample(
        @Value("${val.str:}") String strVal,
        @Value("${val.int:0}") int intValue,
        @Value("${val.boolean:false}") boolean booleanVal,
        @Value("${val.float:0.0}") float floatVal) {
        ...
    }
}
```

The property values are expressed using string values that can be syntactically converted to the type of the target variable.

```
$ java -jar target/appconfig-valueinject-example-*-SNAPSHOT-bootexec.jar \
--app.audience="Command line option" \
--val.str=aString \
--val.int=123 \
--val.boolean=true \
--val.float=123.45
...
Application @Bean says Hey Command line option
strVal=aString
intValue=123
booleanVal=true
floatVal=123.45
```

54.2. Collection Property Types

We can also express properties as a sequence of values and inject the parsed string into Arrays and Collections.

```
...
private final List<Integer> intList;
private final int[] intArray;
private final Set<Integer> intSet;
```

```

public PropertyExample(...  

    @Value("${val.intList}") List<Integer> intList,  

    @Value("${val.intList}") Set<Integer> intSet,  

    @Value("${val.intList}") int[] intArray) {  

    ...  

    --val.intList=1,2,3,3,3  

    ...  

    intList=[1, 2, 3, 3, 3] ①  

    intSet=[1, 2, 3] ②  

    intArray=[1, 2, 3, 3, 3] ③

```

- ① parsed sequence with duplicates injected into List maintained duplicates
- ② parsed sequence with duplicates injected into Set retained only unique values
- ③ parsed sequence with duplicates injected into Array maintained duplicates

54.3. Custom Delimiters (using Spring SpEL)

We can get a bit more elaborate and define a custom delimiter for the values. However, it requires the use of Spring Expression Language (EL; SpEL) `#{} operator`. (Ref: [A Quick Guide to Spring @Value](#))

```

private final List<Integer> intList;  

private final List<Integer> intListDelimiter;  

public PropertyExample(  

    ...  

    @Value("${val.intList}") List<Integer> intList,  

    @Value("#{${val.intListDelimiter}.split('!')}") List<Integer>  

    intListDelimiter, ②  

    ...  

    --val.intList=1,2,3,3,3 --val.intListDelimiter='1!2!3!3' ①  

    ...  

    intList=[1, 2, 3, 3, 3]  

    intListDelimiter=[1, 2, 3, 3, 3]
    ...

```

- ① sequence is expressed on command line using two different delimiters
- ② `val.intListDelimiter` String is read in from raw property value and segmented at the custom ! character

54.4. Map Property Types

We can also leverage Spring EL to inject property values directly into a Map.

```
private final Map<Integer, String> map;
```

```

public PropertyExample( ...
    @Value("#{${val.map:{}}}") Map<Integer, String> map) { ①
...
--val.map="{0:'a', 1:'b,c,d', 2:'x'}"
...
map={0=a, 1=b,c,d, 2=x}

```

① parsed map injected into Map of specific type using Spring Expression Language (`#{}`) operator

54.5. Map Element

We can also use Spring EL to obtain a specific element from a Map.

```

private final Map<String, String> systemProperties;

public PropertyExample(
...
    @Value("#{${val.map:{0:'',3:''}}[3]}") String mapValue, ①
...
    (no args)
...
    mapValue= ②
...
    --val.map={0:'foo', 2:'bar, baz', 3:'buz'}
...
    mapValue=buz ③
...

```

① Spring EL declared to use Map element with key 3 and default to a Map of 2 elements with key 0 and 3

② With no arguments provided, the default 3: '' value was injected

③ With a map provided, the value 3:'buz' was injected

54.6. System Properties

We can also simply inject Java System Properties into a Map using Spring EL.

```

private final Map<String, String> systemProperties;

public PropertyExample(
...
    @Value("#{systemProperties}") Map<String, String> systemProperties) { ①
...
    System.out.println("systemProperties[user.timezone]:" + systemProperties.get(
    "user.timezone")); ②

```

```
...  
systemProperties[user.timezone]=America/New_York
```

- ① Complete Map of system properties is injected
- ② Single element is accessed and printed

54.7. Property Conversion Errors

An error will be reported and the program will not start if the value provided cannot be syntactically converted to the target variable type.

```
$ java -jar target/appconfig-valueinject-example-*-SNAPSHOT-bootexec.jar \  
--val.int=abc  
...  
TypeMismatchException: Failed to convert value of type 'java.lang.String'  
to required type 'int'; nested exception is java.lang.NumberFormatException:  
For input string: "abc"
```

Chapter 55. Summary

In this section we

- defined a value injection for an attribute within a Spring Bean using
 - field injection
 - constructor injection
- defined a default value to use in the event a value is not provided
- defined a specific value to inject at runtime using a command line parameter
- implemented property injection for attributes of different types
 - Built-in types (String, int, boolean, etc)
 - Collection types
 - Maps
- Defined custom parsing techniques using Spring Expression Language (EL)

In future sections we will look to specify properties using aggregate property sources like file(s) rather than specifying each property individually.

Property Source

copyright © 2024 jim stafford (jim.stafford@jhu.edu)

Chapter 56. Introduction

In the previous section we defined a value injection into an attribute of a Spring Bean class and defined a few ways to inject a value on an individual basis. Next, we will set up ways to specify entire collection of property values through files.

56.1. Goals

The student will learn:

- to supply groups of properties using files
- to configure a Spring Boot application using property files
- to flexibly configure and control configurations applied

56.2. Objectives

At the conclusion of this lecture and related exercises, the student will be able to:

1. configure a Spring Boot application using a property file
2. specify a property file for a basename
3. specify a property file packaged within a JAR file
4. specify a property file located on the file system
5. specify both `properties` and `YAML` property file sources
6. specify multiple files to derive an injected property from
7. specify properties based on an active profile
8. specify properties based on placeholder values
9. specify property overrides using different external source options

Chapter 57. Property File Source(s)

Spring Boot uses three key properties when looking for configuration files (Ref: docs.spring.io):

1. `spring.config.name` — one or more base names separated by commas. The default is `application` and the suffixes searched for are `.properties` and `.yml` (or `.yaml`)
2. `spring.profiles.active` — one or more profile names separated by commas used in this context to identify which form of the base name to use. The default is `default` and this value is located at the end of the base filename separated by a dash (-; e.g., `application-default`)
3. `spring.config.location` — one or more directories/packages to search for configuration files or explicit references to specific files. The default is:
 - a. `file:config/` - within a `config` directory in the current directory
 - b. `file:./` - within the current directory
 - c. `classpath:/config/` - within a `config` package in the classpath
 - d. `classpath:/` — within the root package of the classpath

Names are primarily used to identify the base name of the application (e.g., `application` or `myapp`) or of distinct areas (e.g., `database`, `security`). Profiles are primarily used to supply variants of property values. Location is primarily used to identify the search paths to look for configuration files but can be used to override names and profiles when a complete file path is supplied.

57.1. Property File Source Example

In this initial example I will demonstrate `spring.config.name` and `spring.config.location` and use a single value injection similar to previous examples.

Value Injection Target

```
//AppCommand.java
...
    @Value("${app.audience}")
    private String audience;
...
```

However, the source of the property value will not come from the command line. It will come from one of the following property and/or YAML files in our module.

Source Tree

```
src
`-- main
|   |-- java
|   |   '-- ...
|   '-- resources
|       |-- alternate_source.properties
|       |-- alternate_source.yml
```

```
|-- application.properties  
`-- property_source.properties
```

JAR File

```
$ jar tf target/appconfig-propertysource-example-*-SNAPSHOT-bootexec.jar | \  
egrep 'classes.*(properties|yml)'  
BOOT-INF/classes/alternate_source.properties  
BOOT-INF/classes/alternate_source.yml  
BOOT-INF/classes/property_source.properties  
BOOT-INF/classes/application.properties
```

57.2. Example Property File Contents

The four files each declare the same property `app.audience` but with a different value. Spring Boot primarily supports the two file types shown (`properties` and `YAML`). There is [some support for JSON](#) and `XML` is primarily used to define configurations.

The first three below are in [properties format](#).

```
#property_source.properties  
app.audience=Property Source value
```

```
#alternate_source.properties  
app.audience=alternate source property file
```

```
#application.properties  
app.audience=application.properties value
```

This last file is in [YAML format](#).

```
#alternate_source.yml  
app:  
  audience: alternate source YAML file
```

That means the following—which will load the `application.(properties|yml)` file from one of the four locations ...

```
$ java -jar target/appconfig-propertysource-example-*-SNAPSHOT-bootexec.jar  
...  
Application @Bean says Hey application.properties value
```

can also be completed with

```
$ java -jar target/appconfig-propertysource-example-*-SNAPSHOT-bootexec.jar \
--spring.config.location="classpath:/"
...
Application @Bean says Hey application.properties value
```

```
$ java -jar target/appconfig-propertysource-example-*-SNAPSHOT-bootexec.jar \
--spring.config.location="file:src/main/resources/"
...
Application @Bean says Hey application.properties value
```

```
$ java -jar target/appconfig-propertysource-example-*-SNAPSHOT-bootexec.jar \
--spring.config.location="file:src/main/resources/application.properties"
...
Application @Bean says Hey application.properties value
```

```
$ cp src/main/resources/application.properties /tmp/xyz.properties
$ java -jar target/appconfig-propertysource-example-*-SNAPSHOT-bootexec.jar \
--spring.config.name=xyz --spring.config.location="file:/tmp/"
...
Application @Bean says Hey application.properties value
```

57.3. Non-existent Path

If you supply a non-existent path, Spring will report that as an error.

Location Directories Must Exist

```
java -jar target/appconfig-propertysource-example-*-SNAPSHOT-bootexec.jar \
--spring.config.location="file:src/main/resources/,file:src/main/resources/does_not_ex
it/"

[main] ERROR org.springframework.boot.diagnostics.LoggingFailureAnalysisReporter --
*****
APPLICATION FAILED TO START
*****
```

Description:

Config data location 'file:src/main/resources/does_not_exit/' does not exist

Action:

Check that the value 'file:src/main/resources/does_not_exit/' is correct, or prefix it

```
with 'optional':'
```

You can mark the location with `optional:` for cases where it is legitimate for the location not to exist.

Making Location Directory Optional

```
java -jar target/appconfig-propertysource-example-*-.jar \
--spring.config.location="file:src/main/resources/,optional:file:src/main/resources/does_not_exit/"
```

57.4. Path not Ending with Slash ("")

If you supply a path not ending with a slash (""), Spring will also report an error.

```
java -jar target/appconfig-propertysource-example-*-.jar \
--spring.config.location="file:src/main/resources"
...
14:28:23.544 [main] ERROR org.springframework.boot.SpringApplication - Application run failed
java.lang.IllegalStateException: Unable to load config data from
'file:src/main/resources'
...
Caused by: java.lang.IllegalStateException: File extension is not known to any
PropertySourceLoader. If the location is meant to reference a directory, it must end
in '/' or File.separator
```

57.5. Alternate File Examples

We can switch to a different set of configuration files by changing the `spring.config.name` or `spring.config.location` so that ...

```
#property_source.properties
app.audience=Property Source value
```

```
#alternate_source.properties
app.audience=alternate source property file
```

```
#alternate_source.yml
app:
  audience: alternate source YAML file
```

can be used to produce

```
$ java -jar target/appconfig-propertysource-example-*-SNAPSHOT-bootexec.jar \
--spring.config.name=property_source
...
Application @Bean says Hey Property Source value
```

```
$ java -jar target/appconfig-propertysource-example-*-SNAPSHOT-bootexec.jar \
--spring.config.name=alternate_source
...
Application @Bean says Hey alternate source property file
```

```
$ java -jar target/appconfig-propertysource-example-*-SNAPSHOT-bootexec.jar \
--spring.config.location="classpath:alternate_source.properties,classpath:alternate_so
urce.yml"
...
Application @Bean says Hey alternate source YAML file
```

57.6. Series of files

```
#property_source.properties
app.audience=Property Source value
```

```
#alternate_source.properties
app.audience=alternate source property file
```

The default priority is last specified.

```
$ java -jar target/appconfig-propertysource-example-*-SNAPSHOT-bootexec.jar \
--spring.config.name="property_source,alternate_source"
...
Application @Bean says Hey alternate source property file
```

```
$ java -jar target/appconfig-propertysource-example-*-SNAPSHOT-bootexec.jar \
--spring.config.name="alternate_source,property_source"
...
Application @Bean says Hey Property Source value
```

Chapter 58. @PropertySource Annotation

We can define a property to explicitly be loaded using a Spring-provided `@PropertySource` annotation. This annotation can be used on any class that is used as a `@Configuration`, so I will add that to the main application. However, because we are still working with a very simplistic, single property example—I have started a sibling example that only has a single property file so that no priority/overrides from `application.properties` will occur.

Example Source Tree

```
|-- pom.xml
\-- src
  '-- main
    |-- java
    |  '-- info
    |    '-- ejava
    |      '-- examples
    |        '-- app
    |          '-- config
    |            '-- propertysource
    |              '-- annotation
    |                |-- AppCommand.java
    |                '-- PropertySourceApp.java
    '-- resources
      '-- property_source.properties
```

```
#property_source.properties
app.audience=Property Source value
```

Annotation Reference

```
...
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.PropertySource;

@SpringBootApplication
@PropertySource("classpath:property_source.properties") ①
public class PropertySourceApp {
...
}
```

① An explicit reference to the properties file is placed within the annotation on the `@Configuration` class

When we now execute our JAR, we get the contents of the property file.

```
java -jar target/appconfig-propertysource-annotation-example-*-.SNAPSHOT-bootexec.jar
...
```

Application @Bean says Hey Property Source value

We will cover alternate property sources and their priority later within this lecture, `@PropertySource` ends up being one or the lowest priority sources. This permits the application to package a set of factory defaults and optionally allow many of the other sources to override.



Place Factory-Default Properties in @PropertySource

`@PropertySource` references make for a convenient location for components to package factory-supplied, low-priority defaults that can be easily overridden.

Chapter 59. Profiles

In addition to `spring.config.name` and `spring.config.location`, there is a third configuration property—`spring.profiles.active`—that Spring uses when configuring an application. Profiles are identified by

`-(profileName)` at the end of the base filename (e.g., `application-site1.properties`, `myapp-site1.properties`)

I am going to create a new example to help explain this.

Profile Example

```
|-- pom.xml
`-- src
  '-- main
    |-- java
    |  '-- info
    |    '-- ejava
    |      '-- examples
    |        '-- app
    |          '-- config
    |            '-- propertysource
    |              '-- profiles
    |                |-- AppCommand.java
    |                '-- PropertySourceApp.java
    '-- resources
      |-- application-default.properties
      |-- application-site1.properties
      |-- application-site2.properties
      '-- application.properties
```

The example uses the default `spring.config.name` of `application` and supplies four property files.

- each of the property files supplies a common property of `app.commonProperty` to help demonstrate priority
- each of the property files supplies a unique property to help identify whether the file was used

```
#application.properties
app.commonProperty=commonProperty from application.properties
app.appProperty=appProperty from application.properties
```

```
#application-default.properties
app.commonProperty=commonProperty from application-default.properties
app.defaultProperty=defaultProperty from application-default.properties
```

```
#application-site1.properties
```

```
app.commonProperty=commonProperty from application-site1.properties  
app.site1Property=site1Property from application-site1.properties
```

```
#application-site2.properties  
app.commonProperty=commonProperty from application-site2.properties  
app.site2Property=site2Property from application-site2.properties
```

The component class defines an attribute for each of the available properties and defines a default value to identify when they have not been supplied.

```
@Component  
public class AppCommand implements CommandLineRunner {  
    @Value("${app.commonProperty:not supplied}")  
    private String commonProperty;  
    @Value("${app.appProperty:not supplied}")  
    private String appProperty;  
    @Value("${app.defaultProperty:not supplied}")  
    private String defaultProperty;  
    @Value("${app.site1Property:not supplied}")  
    private String site1Property;  
    @Value("${app.site2Property:not supplied}")  
    private String site2Property;
```



In all cases (except when using an alternate `spring.config.name`), we will get the `application.properties` loaded. However, it is used at a lower priority than all other sources.

59.1. Default Profile

If we run the program with no profiles active, we enact the `default` profile. `site1` and `site2` profiles are not loaded.

```
$ java -jar target/appconfig-propertysource-profile-example-*-SNAPSHOT-bootexec.jar  
...  
commonProperty=commonProperty from application-default.properties ①  
appProperty=appProperty from application.properties ②  
defaultProperty=defaultProperty from application-default.properties ③  
site1Property=not supplied ④  
site2Property=not supplied
```

① `commonProperty` was set to the value from `default` profile

② `application.properties` was loaded

③ the `default` profile was loaded

④ `site1` and `site2` profiles were not loaded

59.2. Specific Active Profile

If we activate a specific profile (`site1`) the associated file is loaded and the alternate profiles—including `default`—are not loaded.

```
$ java -jar target/appconfig-propertysource-profile-example-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=site1
...
commonProperty=commonProperty from application-site1.properties ①
appProperty=appProperty from application.properties ②
defaultProperty=not supplied ③
site1Property=site1Property from application-site1.properties ④
site2Property=not supplied ③
```

① `commonProperty` was set to the value from `site1` profile

② `application.properties` was loaded

③ `default` and `site2` profiles were not loaded

④ the `site1` profile was loaded

59.3. Multiple Active Profiles

We can activate multiple profiles at the same time. If they define overlapping properties, the later one specified takes priority.

```
$ java -jar target/appconfig-propertysource-profile-example-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=site1,site2 ①
...
commonProperty=commonProperty from application-site2.properties ①
appProperty=appProperty from application.properties ②
defaultProperty=not supplied ③
site1Property=site1Property from application-site1.properties ④
site2Property=site2Property from application-site2.properties ④

$ java -jar target/appconfig-propertysource-profile-example-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=site2,site1 ①
...
commonProperty=commonProperty from application-site1.properties ①
appProperty=appProperty from application.properties ②
defaultProperty=not supplied ③
site1Property=site1Property from application-site1.properties ④
site2Property=site2Property from application-site2.properties ④
```

① `commonProperty` was set to the value from last specified profile

② `application.properties` was loaded

③ the `default` profile was not loaded

④ site1 and site2 profiles were loaded

59.4. No Associated Profile

If there are no associated profiles with a given `spring.config.name`, then none will be loaded.

```
$ java -jar target/appconfig-propertysource-profile-example-*-SNAPSHOT-bootexec.jar \
--spring.config.name=BOGUS --spring.profiles.active=site1 ①
...
commonProperty=not supplied ①
appProperty=not supplied
defaultProperty=not supplied
site1Property=not supplied
site2Property=not supplied
```

① No profiles where loaded for `spring.config.name BOGUS`

Chapter 60. Property Placeholders

We have the ability to build property values using a placeholder that will come from elsewhere. Consider the following example where there is a common pattern to a specific set of URLs that change based on a base URL value.

- `(config_name).properties` would be the candidate to host the following definition

```
security.authn=${security.service.url}/authentications?user=:user  
security.authz=${security.service.url}/authorizations/roles?user=:user
```

- profiles would host the specific value for the placeholder

- `(config_name)-(profileA).properties`

```
security.service.url=http://localhost:8080
```

- `(config_name)-(profileB).properties`

```
security.service.url=https://acme.com
```

- the default value for the placeholder can be declared in the same property file that uses it

```
security.service.url=https://acme.com  
security.authn=${security.service.url}/authentications?user=:user  
security.authz=${security.service.url}/authorizations/roles?user=:user
```

60.1. Placeholder Demonstration

To demonstrate this further, I am going to add three additional property files to the previous example.

```
'-- src  
  '-- main  
    ...  
    '-- resources  
      |-- ...  
      |-- myapp-site1.properties  
      |-- myapp-site2.properties  
      '-- myapp.properties
```

60.2. Placeholder Property Files

```
# myapp.properties
app.commonProperty=commonProperty from myapp.properties ②
app.appProperty="${app.commonProperty}" used by myapp.property ①
```

① defines a placeholder for another property

② defines a default value for the placeholder within this file



Only the `{}$` characters and property name are specific to property placeholders. Quotes ("") within this property value are part of this example and not anything specific to property placeholders in general.

```
# myapp-site1.properties
app.commonProperty=commonProperty from myapp-site1.properties ①
app.site1Property=site1Property from myapp-site1.properties
```

① defines a value for the placeholder

```
# myapp-site2.properties
app.commonProperty=commonProperty from myapp-site2.properties ①
app.site2Property=site2Property from myapp-site2.properties
```

① defines a value for the placeholder

60.3. Placeholder Value Defined Internally

Without any profiles activated, we obtain a value for the placeholder from within `myapp.properties`.

```
$ java -jar target/appconfig-propertysource-profile-example-*~SNAPSHOT-bootexec.jar \
--spring.config.name=myapp
...
commonProperty=commonProperty from myapp.properties
appProperty="commonProperty from myapp.properties" used by myapp.property ①
defaultProperty=not supplied
site1Property=not supplied
site2Property=not supplied
```

① placeholder value coming from default value defined in same `myapp.properties`

60.4. Placeholder Value Defined in Profile

Activating the `site1` profile causes the placeholder value to get defined by `myapp-site1.properties`.

```
$ java -jar target/appconfig-propertiesource-profile-example-*-.jar \
--spring.config.name=myapp --spring.profiles.active=site1
...
commonProperty=commonProperty from myapp-site1.properties
appProperty="commonProperty from myapp-site1.properties" used by myapp.property ①
defaultProperty=not supplied
site1Property=site1Property from myapp-site1.properties
site2Property=not supplied
```

① placeholder value coming from value defined in `myapp-site1.properties`

60.5. Multiple Active Profiles

Multiple profiles can be activated. By default—the last profile specified has the highest priority.

```
$ java -jar target/appconfig-propertiesource-profile-example-*-.jar \
--spring.config.name=myapp --spring.profiles.active=site1,site2
...
commonProperty=commonProperty from myapp-site2.properties
appProperty="commonProperty from myapp-site2.properties" used by myapp.property ①
defaultProperty=not supplied
site1Property=site1Property from myapp-site1.properties
site2Property=site2Property from myapp-site2.properties
```

① placeholder value coming from value defined in last profile—`myapp-site2.properties`

60.6. Mixing Names, Profiles, and Location

Name, profile, and location constructs can play well together as long as location only references a directory path and not a specific file. In the example below, we are defining a non-default name, a non-default profile, and a non-default location to search for the property files.

```
$ java -jar target/appconfig-propertiesource-profile-example-*-.jar \
--spring.config.name=myapp \
--spring.profiles.active=site1 \
--spring.config.location="file:src/main/resources/"
...
commonProperty=commonProperty from myapp-site1.properties
appProperty="commonProperty from myapp-site1.properties" used by myapp.property
defaultProperty=not supplied
site1Property=site1Property from myapp-site1.properties
site2Property=not supplied
```

The above example located the following property files in the filesystem (not classpath)

- `src/main/resources/myapp.properties`

- src/main/resources/myapp-site1.properties

Chapter 61. Other Common Property Sources

Property files and profiles are, by far, the primary way to configure applications in bulk. However, it is helpful to know other ways to supply or override properties external to the application. One specific example is in deployment platforms where you do not control the command line and must provide some key configuration via environment variables or system properties.

Spring.io lists about a dozen sources of [other property sources](#) in priority order. Use their site for the full list. I won't go through all of them, but will cover the ones that have been the most common and helpful for me in low-to-high Spring priority order.

Each example will inject value into a component property that will be printed.

Component with app.audience @Value Injection

```
@Value("${app.audience}")
private String audience;
```

61.1. Property Source

`@PropertySource` is one of the lowest priority sources. In the following example, I am supplying a custom property file and referencing it from a `@Configuration`. The reference will use a `classpath:` reference to use the file from the module JAR.

@Configuration in the Scan Path References the Property Source

```
@Configuration
@PropertySource(name="propertySource",
    value =
"classpath:info/ejava/examples/app/config/propertysource/packaged_propertySource.properties")
public class PropertySourceConfiguration { }
```

```
#packaged_propertySource.properties
app.audience=packaged_propertySource.properties value
```

Since the `@PropertySource` is the lowest priority source of these examples, I will trigger it by supplying a fictitious `spring.config.name` so that no default profiles override it.

Supplying Custom spring.config.name Clears Higher Priority Sources

```
$ java -jar target/appconfig-propertysource-example-6.1.0-SNAPSHOT-bootexec.jar \
--spring.config.name=none
```

Application @Bean says Hey packaged_propertySource.properties value

61.2. application.properties

If we allow the application to use the default "application" config name, the `application.properties` file will take next precedence.

application.properties with app.audience Property

```
src
`-- main
    |-- ...
    '-- resources
        |-- application.properties

#application.properties
app.audience=application.properties value
```

Property File Property Source

```
$ java -jar target/appconfig-propertysource-example-6.1.0-SNAPSHOT-bootexec.jar
```

```
Application @Bean says Hey application.properties value ①
```

① "application.properties value" comes from a property file

61.3. Profiles

The next level of override — as you know — is a profile.

application-example.properties

```
#application-example.properties
app.audience=application-example.properties value
```

Profile Overrides

```
$ java -jar target/appconfig-propertysource-example-6.1.0-SNAPSHOT-bootexec.jar \
--spring.profiles.active=example
```

```
Application @Bean says Hey application-example.properties value
```

61.4. Environment Variables

We can override the property file specification using an environment variable. The environment variable is expressed in ALL_CAPS with an underscore (_) replacing all dots (.). Therefore

`app.audience` is expressed as `APP_AUDIENCE`. The expression of the environment variable in bash is `export APP_AUDIENCE="some value"`.

Environment Variable Source Overrides Property File Source

```
$ (export APP_AUDIENCE=env && \①  
java -jar target/appconfig-propertysource-example-6.1.0-SNAPSHOT-bootexec.jar \  
--spring.profiles.active=example)
```

Application @Bean says Hey env ②

① "env" comes from an environment variable

② environment variable overrides file source and `example` profile source

Express environment variable and command in subshell parens ('()') versus polluting current shell



```
(export APP_AUDIENCE=... && cmd)
```

Use `unset` to remove an environment variable from the current bash shell.



```
unset APP_AUDIENCE
```

Relaxed binding rules prevent environment variables from expressing case-sensitive values



```
logging.level.org.springframework.web.filter.CommonsRequestLoggingFilte  
r=DEBUG
```

61.5. Java System Properties

Next in priority is the Java system property. The example below shows the `-Dapp.audience=...` Java system property overriding the environment variable.

Java System Property - Property Source

```
$ (export APP_AUDIENCE=env &&  
java -jar \  
-Dapp.audience=sys \①  
target/appconfig-propertysource-example-6.1.0-SNAPSHOT-bootexec.jar)
```

Application @Bean says Hey sys ②

① "sys" comes from system property

- ② system property overrides environment variable and file source

61.6. spring.application.json

Next in priority are properties expressed within a JSON document supplied by the `spring.application.json` property. The `spring.application.json` property can be expressed as an environment variable, Java system property, or command line. The specific JSON will be used based on the priority of the source. The properties within the JSON will override whatever we have demonstrated so far.

61.6.1. JSON Expressed as Environment Variable

spring.application.json Expressed as Environment Variable

```
$ (export APP_AUDIENCE=env && \
SPRING_APPLICATION_JSON='{"app.audience":"envjson"}' && \
java -jar -Dapp.audience=sys \
target/appconfig-propertysource-example-6.1.0-SNAPSHOT-bootexec.jar)
```

Application @Bean says Hey envjson ②

① "envjson" comes from `spring.application.json` property expressed using environment variable

② `spring.application.json` overrides system property, environment variable and file

61.6.2. JSON Expressed as System Property

spring.application.json Expressed as System Property

```
(export APP_AUDIENCE=env && \
SPRING_APPLICATION_JSON='{"app.audience":"envjson"}' && \
java -jar \
-Dapp.audience=sys \
-Dspring.application.json='{"app.audience":"sysjson"}' &① \
target/appconfig-propertysource-example-6.1.0-SNAPSHOT-bootexec.jar)
```

Application @Bean says Hey sysjson ②

① "sysjson" comes from `spring.application.json` property expressed using system property

② system property expression overrides environment variable expression

61.7. Command Line Arguments

Next in priority are command line arguments. The example below shows the `--app.audience=...` command line argument overriding everything we have shown defined to date.

Command Line Argument Property Source

```
$ (export APP_AUDIENCE=env && \
```

```
SPRING_APPLICATION_JSON='{"app.audience":"envjson"}' && \
java -jar \
-Dapp.audience=sys \
-Dspring.application.json='{"app.audience":"sysjson"}' \
target/appconfig-propertysource-example-6.1.0-SNAPSHOT-bootexec.jar \
--app.audience=cmdarg) ①
```

```
Application @Bean says Hey cmdarg ②
```

① "cmdarg" comes from command line argument

② command line argument overrides `spring.application.json`, system property, environment variable and file

61.8. @SpringBootTest.properties

We will soon discuss testing, but know now that properties expressed as part of the `@SpringBootTest` declaration overrides all other property sources.

SpringBootTest Property Source

```
@SpringBootTest(
    properties={"app.audience=test"})
public class SampleNTest {
```

Chapter 62. Summary

In this module we

- supplied property value(s) through a set of property files
- used both `properties` and `YAML` formatted files to express property values
- specified base filename(s) to use using the `--spring.config.name` property
- specified profile(s) to use using the `--spring.profiles.active` property
- specified paths(s) to search using the `--spring.config.location` property
- specified a custom file to load using the `@PropertySource` annotation
- specified multiple names, profiles, and locations
- specified property overrides through multiple types of external sources

In future modules we will show how to leverage these property sources in a way that can make configuring the Java code easier.

Configuration Properties

copyright © 2024 jim stafford (jim.stafford@jhu.edu)

Chapter 63. Introduction

In the previous chapter we mapped properties from different sources and then mapped them directly into individual component Java class attributes. That showed a lot of power but had at least one flaw—each component would define its own injection of a property. If we changed the structure of a property, we would have many places to update and some of that might not be within our code base.

In this chapter we are going to continue to leverage the same property source(s) as before but remove the direct `@Value` injection from the component classes and encapsulate them within a configuration class that gets instantiated, populated, and injected into the component at runtime.

We will also explore adding validation of properties and leveraging tooling to automatically generate boilerplate JavaBean constructs.

63.1. Goals

The student will learn to:

- map a Java `@ConfigurationProperties` class to properties
- define validation rules for property values
- leverage tooling to generate boilerplate code for JavaBean classes
- solve more complex property mapping scenarios
- solve injection mapping or ambiguity

63.2. Objectives

At the conclusion of this lecture and related exercises, the student will be able to:

1. map a Java `@ConfigurationProperties` class to a group of properties
 - generate property metadata — used by IDEs for property editors
2. create read-only `@ConfigurationProperties` class using constructor binding
3. define Jakarta EE Java validation rule for property and have validated at runtime
4. generate boilerplate JavaBean methods using Lombok library
5. use relaxed binding to map between JavaBean and property syntax
6. map nested properties to a `@ConfigurationProperties` class
7. map array properties to a `@ConfigurationProperties` class
8. reuse `@ConfigurationProperties` class to map multiple property trees
9. use `@Qualifier` annotation and other techniques to map or disambiguate an injection

Chapter 64. Mapping properties to @ConfigurationProperties class

Starting off simple, we define a property (`app.config.car.name`) in `application.properties` to hold the name of a car.

```
# application.properties  
app.config.car.name=Suburban
```

64.1. Mapped Java Class

At this point we now want to create a Java class to be instantiated and be assigned the value(s) from the various property sources—`application.properties` in this case, but as we have seen from earlier lectures properties can come from many places. The class follows standard `JavaBean` characteristics

- default constructor to instantiate the class in a default state
- "setter"/"getter" methods to set and get the state of the instance

A `"toString()"` method was also added to self-describe the state of the instance.

```
import org.springframework.boot.context.properties.ConfigurationProperties;  
  
{@ConfigurationProperties("app.config.car") ③  
public class CarProperties { ①  
    private String name;  
  
    //default ctor ②  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name; ②  
    }  
  
    @Override  
    public String toString() {  
        return "CarProperties{name='" + name + "'}";  
    }  
}
```

① class is a standard Java bean with one property

② class designed for us to use its default constructor and a `setter()` to assign value(s)

- ③ class annotated with `@ConfigurationProperties` to identify that is mapped to properties and the property prefix that pertains to this class

64.2. Injection Point

We can have Spring instantiate the bean, set the state, and inject that into a component at runtime and have the state of the bean accessible to the component.

```
...
@Component
public class AppCommand implements CommandLineRunner {
    @Autowired
    private CarProperties carProperties; ①

    public void run(String... args) throws Exception {
        System.out.println("carProperties=" + carProperties); ②
    }
}
```

① Our `@ConfigurationProperties` instance is being injected into a `@Component` class using FIELD injection

② Simple print statement of bean's `toString()` result

64.3. Initial Error

However, if we build and run our application at this point, our injection will fail because Spring was not able to locate what it needed to complete the injection.

```
*****
APPLICATION FAILED TO START
*****
```

Description:

Field `carProperties` in `info.ejava.examples.app.config.configproperties.AppCommand` required a bean of type 'info.ejava.examples.app.config.configproperties.properties.CarProperties' that could not be found.

The injection point has the following annotations:

- `@org.springframework.beans.factory.annotation.Autowired(required=true)`

Action:

Consider defining a bean of type 'info.ejava.examples.app.config.configproperties.properties.CarProperties' in your configuration. ①

- ① Error message indicates that Spring is not seeing our `@ConfigurationProperties` class

64.4. Registering the `@ConfigurationProperties` class

We currently have a similar problem that we had when we implemented our first `@Configuration` and `@Component` classes—the bean was not being scanned. Even though we have our `@ConfigurationProperties` class in the same basic classpath as the `@Configuration` and `@Component` classes—we need a little more to have it processed by Spring. There are several ways to do that:

Example Tree Structure showing Java Package Hierarchy

```
-- java
  '-- info
    '-- ejava
      '-- examples
        '-- app
          '-- config
            '-- configproperties
              |-- AppCommand.java
              |-- ConfigurationPropertiesApp.java ①
              '-- properties
                '-- CarProperties.java ①
-- resources
  '-- application.properties
```

- ① `...properties.CarProperties` Java package is under main class' Java package scope

64.4.1. way 1 - Register Class as a `@Component`

Our package is being scanned by Spring for components, so if we add a `@Component` annotation the `@ConfigurationProperties` class will be automatically picked up.

Example using Component Scan to Trigger `@ConfigurationProperties` processing

```
package info.ejava.examples.app.config.configproperties.properties;
...
@Component
@ConfigurationProperties("app.config.car") ①
public class CarProperties {
```

- ① causes Spring to process the bean and annotation as part of component classpath scanning

- benefits: simple
- drawbacks: harder to override when configuration class and component class are in the same Java class package tree

64.4.2. way 2 - Explicitly Register Class

Explicitly register the class using `@EnableConfigurationProperties` annotation on a `@Configuration`

class (such as the `@SpringBootApplication` class)

Example using `@EnableConfigurationProperties` Scan for Explicit Class

```
import info.ejava.examples.app.config.configproperties.properties.CarProperties;
import org.springframework.boot.context.properties.ConfigurationPropertiesScan;
...
@SpringBootApplication
@EnableConfigurationProperties(CarProperties.class) ①
public class ConfigurationPropertiesApp {
```

① targets a specific `@ConfigurationProperties` class to process

- benefits: `@Configuration` class has explicit control over which configuration properties classes to activate
- drawbacks: application could be coupled with the details if where configurations come from

64.4.3. way 3 - Enable Package Scanning

Enable package scanning for `@ConfigurationProperties` classes with the `@ConfigurationPropertiesScan` annotation

Example using General `@EnableConfigurationProperties` Scan

```
@SpringBootApplication
@ConfigurationPropertiesScan ①
public class ConfigurationPropertiesApp {
```

① allows a generalized scan to be defined that is separate for configurations



We can control which root-level Java packages to scan. The default root is where annotation declared.

- benefits: easy to add more configuration classes without changing application
- drawbacks: generalized scan may accidentally pick up an unwanted configuration

64.4.4. way 4 - Use `@Bean` factory

Create a `@Bean` factory method in a `@Configuration` class for the type .

Example using `@Configuration @Bean Factory`

```
@SpringBootApplication
public class ConfigurationPropertiesApp {
...
@Bean
@ConfigurationProperties("app.config.car") ①
public CarProperties carProperties() {
    return new CarProperties();
```

```
}
```

① gives more control over the runtime mapping of the bean to the `@Configuration` class

- benefits: decouples the `@ConfigurationProperties` class from the specific property prefix used to populate it. This allows for reuse of the same `@ConfigurationProperties` class for multiple prefixes
- drawbacks: implementation spread out between the `@ConfigurationProperties` and `@Configuration` classes. It also prohibits the use of read-only instances since the returned object is not yet populated

For our solution in this example, I am going to use `@ConfigurationPropertiesScan` ("way3") and drop multiple `@ConfigurationProperties` classes into the same classpath and have them automatically scanned for.

64.5. Result

Having things properly in place, we get the instantiated and initialized `CarProperties` `@ConfigurationProperties` class injected into our component(s). Our example `AppCommand` component simply prints the `toString()` result of the instance and we see the property we set in the `applications.property` file.

Property Definition

```
# application.properties
app.config.car.name=Suburban
```

Injected @Component Processing the Bean

```
...
@Component
public class AppCommand implements CommandLineRunner {
    @Autowired
    private CarProperties carProperties;

    public void run(String... args) throws Exception {
        System.out.println("carProperties=" + carProperties);
    }
}
```

Produced Output

```
$ java -jar target/appconfig-configproperties-example-*-SNAPSHOT-bootexec.jar
...
carProperties=CarProperties{name='Suburban'}
```

Chapter 65. Metadata

IDEs have support for linking Java properties to their `@ConfigurationProperty` class information.

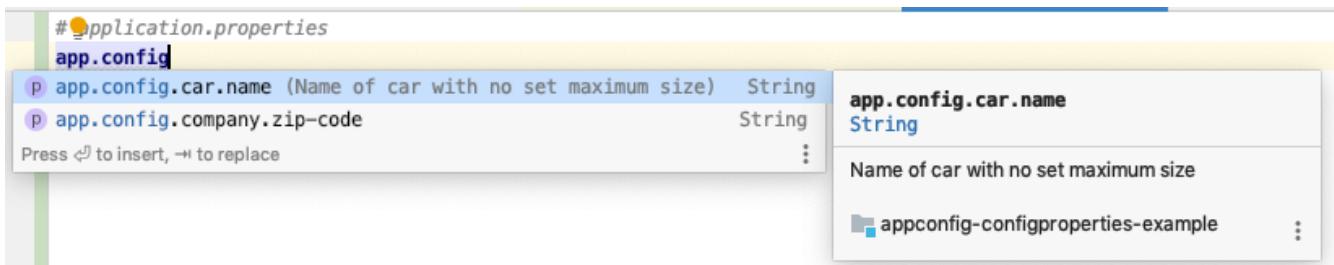


Figure 11. IDE Configuration Property Support

This allows the property editor to know:

- there is a property `app.config.carname`
- any provided Javadoc



Spring Configuration Metadata and IDE support is very helpful when faced with configuring dozens of components with hundreds of properties (or more!)

65.1. Spring Configuration Metadata

IDEs rely on a JSON-formatted metadata file located in `META-INF/spring-configuration-metadata.json` to provide that information.

META-INF/spring-configuration-metadata.json Snippet

```
...
"properties": [
  {
    "name": "app.config.car.name",
    "type": "java.lang.String",
    "description": "Name of car with no set maximum size",
    "sourceType": "info.ejava.examples.app.config.configproperties.properties.CarProperties"
  }
]
...
```

We can author it manually. However, there are ways to automate this.

65.2. Spring Configuration Processor

To have Maven automatically generate the JSON metadata file, add the following dependency to the project to have additional artifacts generated during Java compilation. The Java compiler will inspect and recognize a type of class inside the dependency and call it to perform additional processing. Make it `optional=true` since it is only needed during compilation and not at runtime.

```
<!-- pom.xml dependencies -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-configuration-processor</artifactId> ①
    <optional>true</optional> ②
</dependency>
```

① dependency will generate additional artifacts during compilation

② dependency not required at runtime and can be eliminated from dependents



Dependencies labelled `optional=true` or `scope=provided` are not included in the Spring Boot executable JAR or transitive dependencies in downstream deployments without further configuration by downstream dependents.

65.3. Javadoc Supported

As noted earlier, the metadata also supports documentation extracted from Javadoc comments. To demonstrate this, I will add some simple Javadoc to our example property.

```
@ConfigurationProperties("app.config.car")
public class CarProperties {
    /**
     * Name of car with no set maximum size ①
     */
    private String name;
```

① Javadoc information is extracted from the class and placed in the property metadata

65.4. Rebuild Module

Rebuilding the module with Maven and reloading the module within the IDE should give the IDE additional information it needs to help fill out the properties file.

Metadata File Created During Compilation

```
$ mvn clean compile
```

Produced Metadata File in target/classes Tree

```
target/classes/META-INF/
`-- spring-configuration-metadata.json
```

Produced Metadata File Contents

```
{
  "groups": [
```

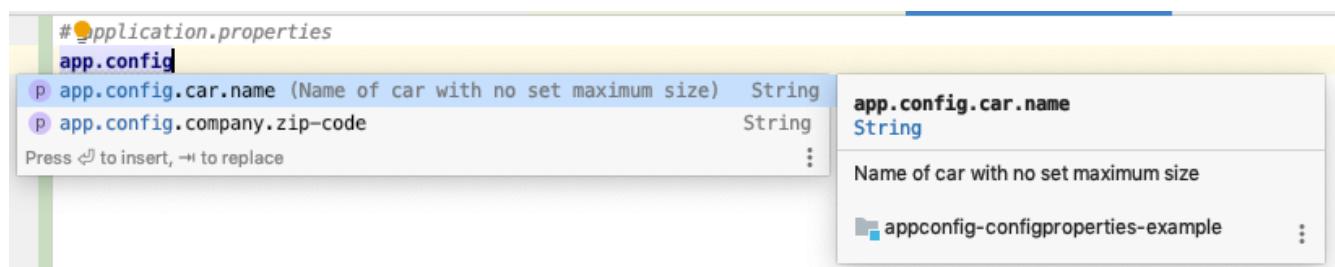
```

{
  "name": "app.config.car",
  "type": "info.ejava.examples.app.config.configproperties.properties.CarProperties",
  "sourceType": "info.ejava.examples.app.config.configproperties.properties.CarProperties"
}
],
"properties": [
{
  "name": "app.config.car.name",
  "type": "java.lang.String",
  "description": "Name of car with no set maximum size",
  "sourceType": "info.ejava.examples.app.config.configproperties.properties.CarProperties"
}
],
"hints": []
}

```

65.5. IDE Property Help

If your IDE supports Spring Boot and property metadata, the property editor will offer help filling out properties.



IntelliJ free Community Edition does not support this feature. The following [link](#) provides a comparison with the for-cost Ultimate Edition.

Chapter 66. Constructor Binding

The previous example was a good start. However, I want to create a slight improvement at this point with a similar example and make the JavaBean read-only. This better depicts the contract we have with properties. They are read-only.

To accomplish a read-only JavaBean, we should remove the setter(s), create a custom constructor that will initialize the attributes at instantiation time, and ideally declare the attributes as final to enforce that they get initialized during construction and never changed.

Spring will automatically use the constructor in this case when there is only one. Add the `@ConstructorBinding` annotation to one of the constructors when there is more than one to choose.

Constructor Binding Example

```
...
import org.springframework.boot.context.properties.bind.ConstructorBinding;

@ConfigurationProperties("app.config.boat")
public class BoatProperties {
    private final String name; ③

    @ConstructorBinding //only required for multiple constructors ②
    public BoatProperties(String name) {
        this.name = name;
    }
    //not used for ConfigurationProperties initialization
    public BoatProperties() { this.name = "default"; }

    //no setter method(s) in read-only example ①
    public String getName() {
        return name;
    }
    @Override
    public String toString() {
        return "BoatProperties{name='" + name + "'}";
    }
}
```

① remove setter methods to better advertise the read-only contract of the bean

② add custom constructor and annotate with `@ConstructorBinding` when multiple ctors

③ make attributes final to better enforce the read-only nature of the bean



`@ConstructorBinding` annotation required on the constructor method when more than one constructor is supplied.

66.1. Property Names Bound to Constructor Parameter Names

When using constructor binding, we no longer have the name of the setter method(s) to help map the properties. The parameter name(s) of the constructor are used instead to resolve the property values.

In the following example, the property `app.config.boat.name` matches the constructor parameter `name`. The result is that we get the output we expect.

```
# application.properties  
app.config.boat.name=Maxum
```

Result of Parameter Name Matching Property Name

```
$ java -jar target/appconfig-configproperties-example-*-SNAPSHOT-bootexec.jar  
...  
boatProperties=BoatProperties{name='Maxum'}
```

66.2. Constructor Parameter Name Mismatch

If we change the constructor parameter name to not match the property name, we will get a null for the property.

```
@ConfigurationProperties("app.config.boat")  
public class BoatProperties {  
    private final String name;  
  
    @ConstructorBinding  
    public BoatProperties(String nameX) { ①  
        this.name = nameX;  
    }  
}
```

- ① constructor argument name has been changed to not match the property name from `application.properties`

Result of Parameter Name not Matching Property Name

```
$ java -jar target/appconfig-configproperties-example-*-SNAPSHOT-bootexec.jar  
...  
boatProperties=BoatProperties{name='null'}
```



We will discuss relaxed binding soon and see that some syntactical differences between the property name and JavaBean property name are accounted for during `@ConfigurationProperties` binding. However, this was a clear case of a name

mis-match that will not be mapped.

Chapter 67. Validation

The error in the last example would have occurred whether we used constructor or setter-based binding. We would have had a possibly vague problem if the property was needed by the application. We can help detect invalid property values for both the setter and constructor approaches by leveraging validation.

[Java validation](#) is a JavaEE/ [Jakarta EE](#) standard API for expressing validation for JavaBeans. It allows us to express constraints on JavaBeans to help further modularize objects within our application.

To add validation to our application, we start by adding the Spring Boot validation starter ([spring-boot-starter-validation](#)) to our pom.xml.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

This will bring in three (3) dependencies

- jakarta.validation-api - this is the validation API and is required to compile the module
- hibernate-validator - this is a validation implementation
- tomcat-embed-el - this is required when expressing validations using [regular expressions](#) with [@Pattern annotation](#)

67.1. Validation Annotations

We trigger Spring to validate our JavaBean when instantiated by the container by adding the [Spring @Validated](#) annotation to the class. We further define the Java attribute with the Jakarta EE [@NotBlank](#) constraint to report an error if the property is ever null or lacks a non-whitespace character.

@ConfigurationProperties JavaBean with Validation

```
...
import org.springframework.validation.annotation.Validated;
import jakarta.validation.constraints.NotBlank;

@ConfigurationProperties("app.config.boat")
@Validated ①
public class BoatProperties {
    @NotBlank ②
    private final String name;

    @ConstructorBinding
    public BoatProperties(String nameX) {
```

```
    this.name = nameX;  
}  
...
```

- ① The Spring `@Validated` annotation tells Spring to validate instances of this class
- ② The Jakarta EE `@NotBlank` annotation tells the validator this field is not allowed to be null or lacking a non-whitespace character



You can locate other validation constraints in the [Validation API](#) and also extend the API to provide more customized validations using the [Validation Spec](#), [Hibernate Validator Documentation](#), or various web searches.

67.2. Validation Error

The error produced is caught by Spring Boot and turned into a helpful description of the problem clearly stating there is a problem with one of the properties specified (when actually it was a problem with the way the JavaBean class was implemented)

```
$ java -jar target/appconfig-configproperties-example-*-SNAPSHOT-bootexec.jar \  
--app.config.boat.name=  
*****  
APPLICATION FAILED TO START  
*****  
Description:  
  
Binding to target  
info.ejava.examples.app.config.configproperties.properties.BoatProperties failed:  
  
Property: app.config.boat.name  
Value: ""  
Origin: "app.config.boat.name" from property source "commandLineArgs"  
Reason: must not be blank  
  
Action:  
  
Update your application's configuration
```



Notice how the error message output by Spring Boot automatically knew what a validation error was and that the invalid property mapped to a specific property name. That is an example of Spring Boot's [FailureAnalyzer](#) framework in action—which aims to make meaningful messages out of what would otherwise be a clunky stack trace.

Chapter 68. Boilerplate JavaBean Methods

Before our implementations gets more complicated, we need to address a simplification we can make to our JavaBean source code which will make all future JavaBean implementations incredibly easy.

Notice all the boilerplate constructor, getter/setter, `toString()`, etc. methods within our earlier JavaBean classes? These methods are primarily based off the attributes of the class. They are commonly implemented by IDEs during development but then become part of the overall code base that has to be maintained over the lifetime of the class. This will only get worse as we add additional attributes to the class when our code gets more complex.

```
...
@ConfigurationProperties("app.config.boat")
@Validated
public class BoatProperties {
    @NotBlank
    private final String name;

    public BoatProperties(String name) { //boilerplate ①
        this.name = name;
    }

    public String getName() { //boilerplate ①
        return name;
    }

    @Override
    public String toString() { //boilerplate ①
        return "BoatProperties{name='" + name + "'}";
    }
}
```

① Many boilerplate methods in source code — likely generated by IDE

68.1. Generating Boilerplate Methods with Lombok

These boilerplate methods can be automatically provided for us at compilation using the [Lombok](#) library. Lombok is not unique to Spring Boot but has been adopted into Spring Boot's overall opinionated approach to developing software and has been integrated into the popular Java IDEs.

I will introduce various Lombok features during later portions of the course and start with a simple case here where all defaults for a JavaBean are desired. The simple Lombok `@Data` annotation intelligently inspects the JavaBean class with just an attribute and supplies boilerplate constructs commonly supplied by the IDE:

- constructor to initialize attributes
- getter

- `toString()`
- `hashCode()` and `equals()`

A setter was not defined by Lombok because the `name` attribute is declared final.

Java Bean using Lombok

```
...
import lombok.Data;

@ConfigurationProperties("app.config.company")
@Data ①
@Validated
public class CompanyProperties {
    @NotNull
    private final String name;
    //constructor ①
    //getter ①
    //toString ①
    //hashCode and equals ①
}
```

① Lombok `@Data` annotation generated constructor, getter(/setter), `toString`, `hashCode`, and `equals`

68.2. Visible Generated Constructs

The additional methods can be identified in a class structure view of an IDE or using Java disassembler (`javap`) command

Example IDE Class Structure View

```
1 package info.ejava.examples.app.config.configproperties.properties;
2
3 import lombok.Data;
4 import org.springframework.boot.context.properties.ConfigurationProperties;
5 import org.springframework.boot.context.properties.ConstructorBinding;
6 import org.springframework.validation.annotation.Validated;
7
8 import javax.validation.constraints.NotNull;
9
10 /**
11  * This class provides a example of ConfigurationProperties class that uses ...
12 */
13 @ConfigurationProperties("app.config.company")
14 @ConstructorBinding
15 @Data
16 @Validated
17 public class CompanyProperties {
18     @NotNull
19     private final String name;
20 }
21
```



You may need to locate a compiler option within your IDE properties to make the code generation within your IDE.

javap Class Structure Output

```
$ javap -cp target/classes
```

```

info.ejava.examples.app.config.configproperties.properties.CompanyProperties
Compiled from "CompanyProperties.java"
public class
info.ejava.examples.app.config.configproperties.properties.CompanyProperties {
    public
info.ejava.examples.app.config.configproperties.properties.CompanyProperties(java.lang
.String);
    public java.lang.String getName();
    public boolean equals(java.lang.Object);
    protected boolean canEqual(java.lang.Object);
    public int hashCode();
    public java.lang.String toString();
}

```

68.3. Lombok Build Dependency

The Lombok annotations are defined with `RetentionPolicy.SOURCE`. That means they are discarded by the compiler and not available at runtime.

Lombok Annotations are only used at Compile-time

```

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.SOURCE)
public @interface Data {

```

That permits us to declare the dependency as `scope=provided` to eliminate it from the application's executable JAR and transitive dependencies and have no extra bloat in the module as well.

Maven Dependency

```

<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <scope>provided</scope>
</dependency>

```

68.4. Example Output

Running our example using the same, simple `toString()` print statement and property definitions produces near identical results from the caller's perspective. The only difference here is the specific text used in the returned string.

```

...
@Autowired
private BoatProperties boatProperties;
@Autowired
private CompanyProperties companyProperties;

```

```
public void run(String... args) throws Exception {
    System.out.println("boatProperties=" + boatProperties); ①
    System.out.println("====");
    System.out.println("companyProperties=" + companyProperties); ②
...
}
```

① `BoatProperties` JavaBean methods were provided by hand

② `CompanyProperties` JavaBean methods were provided by Lombok

```
# application.properties
app.config.boat.name=Maxum
app.config.company.name=Acme
```

```
$ java -jar target/appconfig-configproperties-example-*-SNAPSHOT-bootexec.jar
boatProperties=BoatProperties{name='Maxum'}
=====
companyProperties=CompanyProperties(name=Acme)
```

With very infrequent issues, adding Lombok to our development approach for JavaBeans is almost a 100% win situation. 80-90% of the JavaBean class is written for us and we can override the defaults at any time with further annotations or custom methods. The fact that Lombok will not replace methods we have manually provided for the class always gives us an escape route in the event something needs to be customized.

Chapter 69. Relaxed Binding

One of the key differences between Spring's `@Value` injection and `@ConfigurationProperties` is the support for relaxed binding by the latter. With relaxed binding, property definitions do not have to be an exact match. JavaBean properties are commonly defined with camelCase. Property definitions can come in a number of [different case formats](#). Here is a few.

- camelCase
- UpperCamelCase
- kebab-case
- snake_case
- UPPERCASE

69.1. Relaxed Binding Example JavaBean

In this example, I am going to add a class to express many different properties of a business. Each of the attributes is expressed using camelCase to be consistent with common [Java coding conventions](#) and further validated using Jakarta EE Validation.

JavaBean Attributes using camelCase

```
@ConfigurationProperties("app.config.business")
@Data
@Validated
public class BusinessProperties {
    @NotNull
    private final String name;
    @NotNull
    private final String streetAddress;
    @NotNull
    private final String city;
    @NotNull
    private final String state;
    @NotNull
    private final String zipCode;
    private final String notes;
}
```

69.2. Relaxed Binding Example Properties

The properties supplied provide an example of the relaxed binding Spring implements between property and JavaBean definitions.

Example Properties to Demonstrate Relaxed Binding

```
# application.properties
```

```
app.config.business.name=Acme
app.config.business.street-address=100 Suburban Dr
app.config.business.CITY>Newark
app.config.business.State=DE
app.config.business.zip_code=19711
app.config.business.notess=This is a property name typo
```

- kebab-case `street-address` matched Java camelCase `streetAddress`
- UPPERCASE `CITY` matched Java camelCase `city`
- UpperCamelCase `State` matched Java camelCase `state`
- snake_case `zip_code` matched Java camelCase `zipCode`
- typo `notess` does not match Java camelCase `notes`

69.3. Relaxed Binding Example Output

These relaxed bindings are shown in the following output. However, the `note` attribute is an example that there is no magic when it comes to correcting typo errors. The extra character in `notess` prevented a mapping to the `notes` attribute. The IDE/metadata can help avoid the error and validation can identify when the error exists.

```
$ java -jar target/appconfig-configproperties-example-*-.SNAPSHOT-bootexec.jar
...
businessProperties=BusinessProperties(name=Acme, streetAddress=100 Suburban Dr,
city>Newark, state=DE, zipCode=19711, notes=null)
```

Chapter 70. Nested Properties

The previous examples used a flat property model. That may not always be the case. In this example we will look into mapping nested properties.

Nested Properties Example

```
① app.config.corp.name=Acme  
② app.config.corp.address.street=100 Suburban Dr  
    app.config.corp.address.city>Newark  
    app.config.corp.address.state>DE  
    app.config.corp.address.zip=19711
```

① `name` is part of a flat property model below `corp`

② `address` is a container of nested properties

70.1. Nested Properties JavaBean Mapping

The mapping of the nested class is no surprise. We supply a JavaBean to hold their nested properties and reference it from the host/outer-class.

Nested Property Mapping

```
...  
@Data  
public class AddressProperties {  
    private final String street;  
    @NotNull  
    private final String city;  
    @NotNull  
    private final String state;  
    @NotNull  
    private final String zip;  
}
```

70.2. Nested Properties Host JavaBean Mapping

The host class (`CorporateProperties`) declares the base property prefix and a reference (`address`) to the nested class.

Host Property Mapping

```
...  
import org.springframework.boot.context.properties.NestedConfigurationProperty;  
  
@ConfigurationProperties("app.config.corp")
```

```
@Data  
@Validated  
public class CorporationProperties {  
    @NotNull  
    private final String name;  
    @NestedConfigurationProperty //needed for metadata  
    @NotNull  
    //@Valid  
    private final AddressProperties address;
```



The `@NestedConfigurationProperty` is only supplied to generate correct metadata—otherwise only a single `address` property will be identified to exist within the generated metadata.



The validation initiated by the `@Validated` annotation seems to automatically propagate into the nested `AddressProperties` class without the need to add `@Valid` annotation.

70.3. Nested Properties Output

The defined properties are populated within the host and nested bean and accessible to components within the application.

Nested Property Example Output

```
$ java -jar target/appconfig-configproperties-example-*-SNAPSHOT-bootexec.jar  
...  
corporationProperties=CorporationProperties(name=Acme,  
    address=AddressProperties(street=null, city>Newark, state>DE, zip>19711))
```

Chapter 71. Property Arrays

As the previous example begins to show, property mapping can begin to get complex. I won't demonstrate all of them. Please consult [documentation](#) available on the Internet for a complete view. However, I will demonstrate an initial collection mapping to arrays to get started going a level deeper.

In this example, `RouteProperties` hosts a local `name` property and a list of `stops` that are of type `AddressProperties` that we used before.

Property Array JavaBean Mapping

```
...
@ConfigurationProperties("app.config.route")
@Data
@Validated
public class RouteProperties {
    @NotNull
    private String name;
    @NestedConfigurationProperty
    @NotNull
    @Size(min = 1)
    private List<AddressProperties> stops; ①
...
}
```

① `RouteProperties` hosts list of stops as `AddressProperties`

71.1. Property Arrays Definition

The above can be mapped using a properties format.

Property Arrays Example Properties Definition

```
# application.properties
app.config.route.name: Superbowl
app.config.route.stops[0].street: 1101 Russell St
app.config.route.stops[0].city: Baltimore
app.config.route.stops[0].state: MD
app.config.route.stops[0].zip: 21230
app.config.route.stops[1].street: 347 Don Shula Drive
app.config.route.stops[1].city: Miami
app.config.route.stops[1].state: FLA
app.config.route.stops[1].zip: 33056
```

However, it may be easier to map using [YAML](#).

Property Arrays Example YAML Definition

```
# application.yml
```

```
app:  
  config:  
    route:  
      name: Superbowl  
      stops:  
        - street: 1101 Russell St  
          city: Baltimore  
          state: MD  
          zip: 21230  
        - street: 347 Don Shula Drive  
          city: Miami  
          state: FLA  
          zip: 33056
```

71.2. Property Arrays Output

Injecting that into our application and printing the state of the bean (with a little formatting) produces the following output showing that each of the `stops` were added to the `route` using the `AddressProperty`.

Property Arrays Example Output

```
$ java -jar target/appconfig-configproperties-example-*-SNAPSHOT-bootexec.jar  
...  
routeProperties=RouteProperties(name=Superbowl, stops=[  
  AddressProperties(street=1101 Russell St, city=Baltimore, state=MD, zip=21230),  
  AddressProperties(street=347 Don Shula Drive, city=Miami, state=FLA, zip=33056)  
])
```

Chapter 72. System Properties

Note that Java properties can come from several sources and we are able to map them from standard Java system properties as well.

The following example shows mapping three (3) system properties: `user.name`, `user.home`, and `user.timezone` to a `@ConfigurationProperties` class.

Example System Properties JavaBean

```
@ConfigurationProperties("user")
@Data
public class UserProperties {
    @NotNull
    private final String name; ①
    @NotNull
    private final String home; ②
    @NotNull
    private final String timezone; ③
```

① mapped to SystemProperty `user.name`

② mapped to SystemProperty `user.home`

③ mapped to SystemProperty `user.timezone`

72.1. System Properties Usage

Injecting that into our components give us access to mapped properties and, of course, access to them using standard getters and not just `toString()` output.

Example System Properties Usage

```
@Component
public class AppCommand implements CommandLineRunner {
    ...
    @Autowired
    private UserProperties userProps;

    public void run(String... args) throws Exception {
        ...
        System.out.println(userProps); ①
        System.out.println("user.home=" + userProps.getHome()); ②
```

① output `UserProperties` `toString`

② get specific value mapped from `user.home`

System Properties Example Output

```
$ java -jar target/appconfig-configproperties-example-*~SNAPSHOT-bootexec.jar
```

```
...  
UserProperties(name=jim, home=/Users/jim, timezone=America/New_York)  
user.home=/Users/jim
```

Chapter 73. @ConfigurationProperties Class Reuse

The examples to date have been singleton values mapped to one root source. However, as we saw with [AddressProperties](#), we could have multiple groups of properties with the same structure and different root prefix.

In the following example we have two instances of person. One has the prefix of `owner` and the other `manager`, but they both follow the same structural schema.

Example Properties with Common Structure

```
# application.yml
owner: ①
  name: Steve Bushati
  address:
    city: Millersville
    state: MD
    zip: 21108

manager: ②
  name: Eric Decosta
  address:
    city: Owings Mills
    state: MD
    zip: 21117
```

① `owner` and `manager` root prefixes both follow the same structural schema

73.1. @ConfigurationProperties Class Reuse Mapping

We would like two (2) bean instances that represent their respective person implemented as one JavaBean class. We can structurally map both to the same class and create two instances of that class. However when we do that—we can no longer apply the [@ConfigurationProperties](#) annotation and prefix to the bean class because the prefix will be instance-specific

@ConfigurationProperties Class Reuse JavaBean Mapping

```
//@ConfigurationProperties("???") multiple prefixes mapped ①
@Data
@Validated
public class PersonProperties {
  @NotNull
  private String name;
  @NestedConfigurationProperty
  @NotNull
  private AddressProperties address;
```

① unable to apply root prefix-specific `@ConfigurationProperties` to class

73.2. `@ConfigurationProperties @Bean Factory`

We can solve the issue of having two (2) separate leading prefixes by adding a `@Bean` factory method for each use and we can use our root-level application class to host those factory methods.

@Bean Factory Methods for Separate Property Root Prefixes

```
@SpringBootApplication
@ConfigurationPropertiesScan
public class ConfigurationPropertiesApp {
    ...
    @Bean
    @ConfigurationProperties("owner") ②
    public PersonProperties ownerProps() {
        return new PersonProperties(); ①
    }

    @Bean
    @ConfigurationProperties("manager") ②
    public PersonProperties managerProps() {
        return new PersonProperties(); ①
    }
}
```

① `@Bean` factory method returns JavaBean instance to use

② Spring populates the JavaBean according to the `ConfigurationProperties` annotation



We are no longer able to use read-only JavaBeans when using the `@Bean` factory method in this way. We are returning a default instance for Spring to populate based on the specified `@ConfigurationProperties` prefix of the factory method.

73.3. Injecting `ownerProps`

Taking this one instance at a time, when we inject an instance of `PersonProperties` into the `ownerProps` attribute of our component, the `ownerProps @Bean` factory is called and we get the information for our owner.

Owner Person Injection

```
@Component
public class AppCommand implements CommandLineRunner {
    @Autowired
    private PersonProperties ownerProps;
```

Owner Person Injection Result

```
$ java -jar target/appconfig-configproperties-example-*-.SNAPSHOT-bootexec.jar
```

```
...
PersonProperties(name=Steve Bushati, address=AddressProperties(street=null,
city=Millersville, state=MD, zip=21108))
```

Great! However, there was something subtle there that allowed things to work.

73.4. Injection Matching

Spring had two `@Bean` factory methods to choose from to produce an instance of `PersonProperties`.

Two PersonProperties Sources

```
@Bean
@ConfigurationProperties("owner")
public PersonProperties ownerProps() {
...
@Bean
@ConfigurationProperties("manager")
public PersonProperties managerProps() {
...
}
```

The `ownerProps` `@Bean` factory method name happened to match the `ownerProps` Java attribute name and that resolved the ambiguity.

Target Attribute Name for Injection provides Qualifier

```
@Component
public class AppCommand implements CommandLineRunner {
    @Autowired
    private PersonProperties ownerProps; ①
```

① Attribute name of injected bean matches `@Bean` factory method name

73.5. Ambiguous Injection

If we were to add the `manager` and specifically not make the two names match, there will be ambiguity as to which `@Bean` factory to use. The injected attribute name is `manager` and the desired `@Bean` factory method name is `managerProps`.

Manager Person Injection

```
@Component
public class AppCommand implements CommandLineRunner {
    @Autowired
    private PersonProperties manager; ①
```

① Java attribute name does not match `@Bean` factory method name

```
$ java -jar target/appconfig-configproperties-example-*-SNAPSHOT-bootexec.jar
*****
APPLICATION FAILED TO START
*****
Description:

Field manager in info.ejava.examples.app.config.configproperties.AppCommand
required a single bean, but 2 were found:
- ownerProps: defined by method 'ownerProps' in
  info.ejava.examples.app.config.configproperties.ConfigurationPropertiesApp
- managerProps: defined by method 'managerProps' in
  info.ejava.examples.app.config.configproperties.ConfigurationPropertiesApp
```

This may be due to missing parameter name information

Action:

Consider marking one of the beans as `@Primary`, updating the consumer to accept multiple beans,
or using `@Qualifier` to identify the bean that should be consumed

Ensure that your compiler is configured to use the '`-parameters`' flag.
You may need to update both your build tool settings as well as your IDE.

73.6. Injection `@Qualifier`

As the error message states, we can solve this one of several ways. The `@Qualifier` route is mostly what we want and can do that one of at least three ways.

73.7. way1: Create Custom `@Qualifier` Annotation

Create a custom `@Qualifier` annotation and apply that to the `@Bean` factory and injection point.

- benefits: eliminates string name matching between factory mechanism and attribute
- drawbacks: new annotation must be created and applied to both factory and injection point

Custom @Manager Qualifier Annotation

```
package info.ejava.examples.app.config.configproperties.properties;

import org.springframework.beans.factory.annotation.Qualifier;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Qualifier
```

```

@Target({ElementType.METHOD, ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
public @interface Manager {
}

```

@Manager Annotation Applied to @Bean Factory Method

```

@Bean
@ConfigurationProperties("manager")
@Manager ①
public PersonProperties managerProps() {
    return new PersonProperties();
}

```

① `@Manager` annotation used to add additional qualification beyond just type

@Manager Annotation Applied to Injection Point

```

@Autowired
private PersonProperties ownerProps;
@Autowired
@Manager ①
private PersonProperties manager;

```

① `@Manager` annotation is used to disambiguate the factory choices

73.8. way2: @Bean Factory Method Name as Qualifier

Use the name of the `@Bean` factory method as a qualifier.

- benefits: no custom qualifier class required and factory signature does not need to be modified
- drawbacks: text string must match factory method name

Example using String name of @Bean

```

@Autowired
private PersonProperties ownerProps;
@Autowired
@Qualifier("managerProps") ①
private PersonProperties manager;

```

① `@Bean` factory name is being applied as a qualifier versus defining a type

73.9. way3: Match @Bean Factory Method Name

Change the name of the injected attribute to match the `@Bean` factory method name

- benefits: simple and properly represents the semantics of the singleton property

- drawbacks: injected attribute name must match factory method name

PersonProperties Sources

```

@Bean
@ConfigurationProperties("owner")
public PersonProperties ownerProps() {
...
}

@Bean
@ConfigurationProperties("manager")
public PersonProperties managerProps() {
...
}

```

Injection Points

```

@Autowired
private PersonProperties ownerProps;
@Autowired
private PersonProperties managerProps; ①

```

① Attribute name of injected bean matches `@Bean` factory method name

73.10. Ambiguous Injection Summary

Factory choices and qualifiers is a whole topic within itself. However, this set of examples showed how `@ConfigurationProperties` can leverage `@Bean` factories to assist in additional complex property mappings. We likely will be happy taking the simple `way3` solution but it is good to know there is an easy way to use a `@Qualifier` annotation when we do not want to rely on a textual name match.

Chapter 74. Summary

In this module we

- mapped properties from property sources to JavaBean classes annotated with `@ConfigurationProperties` and injected them into component classes
- generated property metadata that can be used by IDEs to provide an aid to configuring properties
- implemented a read-only JavaBean
- defined property validation using Jakarta EE Java Validation framework
- generated boilerplate JavaBean constructs with the Lombok library
- demonstrated how relaxed binding can lead to more flexible property names
- mapped flat/simple properties, nested properties, and collections of properties
- leveraged custom `@Bean` factories to reuse common property structure for different root instances
- leveraged `@Qualifier`s in order to map or disambiguate injections

Auto Configuration

copyright © 2024 jim stafford (jim.stafford@jhu.edu)

Chapter 75. Introduction

Thus far we have focused on how to configure an application within the primary application module, under fairly static conditions, and applied directly to a single application.

However, our application configuration will likely be required to be:

- **dynamically determined** - Application configurations commonly need to be dynamic based on libraries present, properties defined, resources found, etc. at startup. For example, what database will be used when in development, integration, or production? What security should be enabled in development versus production areas?
- **modularized and not repeated** - Breaking the application down into separate components and making these components reusable in multiple applications by physically breaking them into separate modules is a good practice. However, that leaves us with the repeated responsibility to configure the components reused. Many times there could be dozens of choices to make within a component configuration, and the application can be significantly simplified if an opinionated configuration can be supplied based on the runtime environment of the module.

If you find yourself needing configurations determined dynamically at runtime or find yourself solving a repeated problem and bundling that into a library shared by multiple applications, then you are going to want to master the concepts within Spring Boot's Auto-configuration capability that will be discussed here. Some of these Auto-configuration capabilities mentioned can be placed directly into the application while others are meant to be placed into separate Auto-configuration modules called "starter" modules that can come with an opinionated, default way to configure the component for use with as little work as possible.

75.1. Goals

The student will learn to:

- Enable/disable bean creation based on condition(s) at startup
- Create Auto-configuration/Starter module(s) that establish necessary dependencies and conditionally supplies beans
- Resolve conflicts between alternate configurations
- Locate environment and condition details to debug Auto-configuration issues

75.2. Objectives

At the conclusion of this lecture and related exercises, the student will be able to:

1. Enable a `@Component`, `@Configuration` class, or `@Bean` factory method based on the result of a condition at startup
2. Create Spring Boot Auto-configuration/Starter module(s)
3. Bootstrap Auto-configuration classes into applications using a Spring Boot 3 `org.springframework.boot.autoconfigure.AutoConfiguration.imports` metadata file

4. Create a conditional component based on the presence of a property value
5. Create a conditional component based on a missing component
6. Create a conditional component based on the presence of a class
7. Define a processing dependency order for Auto-configuration classes
8. Access textual debug information relative to conditions using the `debug` property
9. Access web-based debug information relative to conditionals and properties using the Spring Boot Actuator

Ref: [Creating Your Own Auto-configuration](#)

Chapter 76. Review: Configuration Class

As we have seen earlier, `@Configuration` classes are how we bootstrap an application using Java classes. They are the modern alternative to the legacy XML definitions that basically do the same thing—define and configure beans.

`@Configuration` classes can be the `@SpringBootApplication` class itself. This would be appropriate for a small application.

Configuration supplied within `@SpringBootApplication` Class

```
@SpringBootApplication
//==> wraps @EnableAutoConfiguration
//==> wraps @SpringBootConfiguration
//          ==> wraps @Configuration
public class SelfConfiguredApp {
    public static void main(String...args) {
        SpringApplication.run(SelfConfiguredApp.class, args);
    }

    @Bean
    public Hello hello() {
        return new StdOutHello("Application @Bean says Hey");
    }
}
```

76.1. Separate `@Configuration` Class

`@Configuration` classes can be broken out into separate classes. This would be appropriate for larger applications with distinct areas to be configured.

```
@Configuration(proxyBeanMethods = false) ②
public class AConfigurationClass {
    @Bean ①
    public Hello hello() {
        return new StdOutHello("...");
    }
}
```

① bean scope defaults to "singleton"

② nothing directly calling the `@Bean` factory method; establishing a CGLIB proxy is unnecessary



`@Configuration` classes are commonly annotated with the `proxyMethods=false` attribute that tells Spring not to create extra proxy code to enforce normal, singleton return of the created instance to be shared by all callers since `@Configuration` class instances are only called by Spring. The `javadoc` for the annotation attribute describes the extra and unnecessary work saved.

Chapter 77. Conditional Configuration

We can make `@Bean` factory methods (or the `@Component` annotated class) and entire `@Configuration` classes dependent on conditions found at startup. The following example uses the `@ConditionalOnProperty` annotation to define a `Hello` bean based on the presence of the `hello.quiet` property having the value `true`.

Property Condition Example

```
...
import org.springframework.boot.autoconfigure.condition.ConditionalOnProperty;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class StarterConfiguredApp {
    public static void main(String...args) {
        SpringApplication.run(StarterConfiguredApp.class, args);
    }

    @Bean
    @ConditionalOnProperty(prefix="hello", name="quiet", havingValue="true") ①
    public Hello quietHello() {
        return new StdOutHello("(hello.quiet property condition set, Application @Bean
says hi)");
    }
}
```

① `@ConditionalOnProperty` annotation used to define a `Hello` bean based on the presence of the `hello.quiet` property having the value `true`

77.1. Property Value Condition Satisfied

The following is an example of the property being defined with the targeted value.

Property Value Condition Satisfied Result

```
$ java -jar target/appconfig-autoconfig-*-*-SNAPSHOT-bootexec.jar --hello.quiet=true ①
...
(hello.quiet property condition set, Application @Bean says hi) World ②
```

① matching property supplied using command line

② satisfies property condition in `@SpringBootApplication`



The (parentheses) is trying to indicate a *whisper*. `hello.quiet=true` property turns on this behavior.

77.2. Property Value Condition Not Satisfied

The following is an example of when the property is missing. Since there is no `Hello` bean factory, we encounter an error that we will look to solve using a separate Auto-configuration module.

Property Value Condition Not Satisfied

```
$ java -jar target/appconfig-autoconfig-*-SNAPSHOT-bootexec.jar ①
...
*****
APPLICATION FAILED TO START
*****
```

Description:

Parameter 0 of constructor in `info.ejava.examples.app.config.auto.AppCommand` required a bean of type '`info.ejava.examples.app.hello.Hello`' that could not be found. ②

Action:

Consider defining a bean of type '`info.ejava.examples.app.hello.Hello`' in your configuration.

① property either not specified or not specified with targeted value

② property condition within `@SpringBootApplication` not satisfied

Chapter 78. Two Primary Configuration Phases

Configuration processing within Spring Boot is separated into two primary phases:

1. User-defined configuration classes

- processed first
- part of the application module
- located through the use of a `@ComponentScan` (wrapped by `@SpringBootApplication`)
- establish the base configuration for the application
- fill in any fine-tuning details.

2. Auto-configuration classes

- parsed second
- outside the scope of the `@ComponentScan`
- placed in separate modules, identified by metadata within those modules
- enabled by application using `@EnableAutoConfiguration` (also wrapped by `@SpringBootApplication`)
- provide defaults to fill in the reusable parts of the application
- use User-defined configuration for details

Chapter 79. Auto-Configuration

An Auto-configuration class is technically no different from any other `@Configuration` class except that it is inspected after the User-defined `@Configuration` class(es) processing is complete and based on being named in a descriptor file within `META-INF/`. This alternate identification and second pass processing allows the core application to make key directional and detailed decisions and control conditions for the Auto-configuration class(es).

The following Auto-configuration class example defines an **unconditional** `Hello` bean factory configured using a `@ConfigurationProperties` class.

Example Auto-Configuration Class

```
package info.ejava.examples.app.hello; ②
...
@Configuration(proxyBeanMethods = false)
@EnableConfigurationProperties(HelloProperties.class)
public class HelloAutoConfiguration {
    @Bean ①
    public Hello hello(HelloProperties helloProperties) {
        return new StdOutHello(helloProperties.getGreeting());
    }
}
```

- ① Example Auto-configuration class provides **unconditional** `@Bean` factory for `Hello`
- ② this `@Configuration` package is outside the default scanning scope of `@SpringBootApplication`

Auto-Configuration Packages are Separate from Application

Auto-Configuration classes are designed to be outside the scope of the `@SpringBootApplication` package scanning. Otherwise, it would end up being a normal `@Configuration` class and processed within the main application JAR pre-processing.

 package info.ejava.examples.app.config.auto;
@SpringBootApplication

```
package info.ejava.examples.app.hello; ①

@Configuration(proxyBeanMethods = false)
public class HelloAutoConfiguration {
```

- ① `app.hello` is not under `app.config.auto`

79.1. @AutoConfiguration Annotation

Spring Boot 2.7 added the `@AutoConfiguration` annotation, which

- extends `@Configuration`
- permanently sets `proxyBeanMethods` to false
- contains aliases for before/after configuration processing order

```
import org.springframework.context.annotation.Configuration;
import org.springframework.boot.autoconfigure.AutoConfiguration;

//@Configuration(proxyBeanMethods = false)
@AutoConfiguration
public class HelloAutoConfiguration {
```

It helps document the purpose of the `@Configuration` class and provides some ease of use features, but the `@Configuration` annotation can still be used.

79.2. Supporting @ConfigurationProperties

This particular `@Bean` factory defines the `@ConfigurationProperties` class to encapsulate the details of configuring Hello. It supplies a default greeting making it optional for the User-defined configuration to do anything.

Example Auto-Configuration Properties Class

```
@ConfigurationProperties("hello")
@Data
@Validated
public class HelloProperties {
    @NotBlank
    private String greeting = "HelloProperties default greeting says Hola!"; ①
}
```

① Value used if user-configuration does not specify a property value

79.3. Locating Auto Configuration Classes

A dependency JAR makes the Auto-configuration class(es) known to the application by supplying a metadata file (`META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports`) and listing the Auto-configuration classes within that file.

The example below shows the metadata file ("META-INF/...AutoConfiguration.imports") and an Auto-configuration class ("HelloAutoConfiguration") that will be named within that metadata file.

Auto-configuration Module JAR

```
$ jar tf target/hello-starter-*-SNAPSHOT.jar | egrep -v '/$|maven|MANIFEST.MF'  
META-INF/spring-configuration-metadata.json ②  
META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports ①  
info/ejava/examples/app/hello/HelloProperties.class  
info/ejava/examples/app/hello/HelloAutoConfiguration.class
```

① "auto-configuration" dependency JAR supplies … `AutoConfiguration.imports`

② `@ConfigurationProperties` class metadata generated by maven plugin for use by IDEs



It is common best-practice to host Auto-configuration classes in a separate module than the beans it configures. The `Hello` interface and `Hello` implementation(s) comply with this convention and are housed in separate modules.

79.4. META-INF Auto-configuration Metadata File

Auto-configuration classes are registered in the … `AutoConfiguration.imports` file by listing the class' fully qualified name, one per line.

Spring Boot 3 AutoConfiguration Metadata File

```
# src/main/resources/META-  
INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports  
  
info.ejava.examples.app.hello.HelloAutoConfiguration ①  
info.ejava.examples.app.hello.HelloResourceAutoConfiguration ②
```

① Auto-configuration class registration

② this class is part of a later example; multiple classes are listed one-per-line

79.5. Spring Boot 2 META-INF/spring.factories

Prior to Spring Boot 2.7, the general purpose `META-INF/spring.factories` file was used to bootstrap auto-configuration classes. This approach was deprecated in 2.7 and eliminated in Spring Boot 3. If you are ever working with a legacy version of Spring Boot, you will have to use this approach.

Auto-configuration classes were registered using the property name equaling the fully qualified classname of the `@EnableAutoConfiguration` annotation and the value equaling the fully qualified classname of the Auto-configuration class(es). Multiple classes can be specified separated by commas. The last entry on a line cannot end with a comma.

Spring Boot 2 Auto-Configuration Metadata Entry

```
# src/main/resources/META-INF/spring.factories  
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\  
info.ejava.examples.app.hello.HelloAutoConfiguration, \ ①
```

① Auto-configuration class registration



The last line of the property cannot end with a comma or Spring Boot 2 will interpret entry as an empty class name

79.6. Example Auto-Configuration Module Source Tree

Our configuration and properties class—along with the `org.springframework.boot.autoconfigure.AutoConfiguration.imports` file get placed in a separate module source tree.

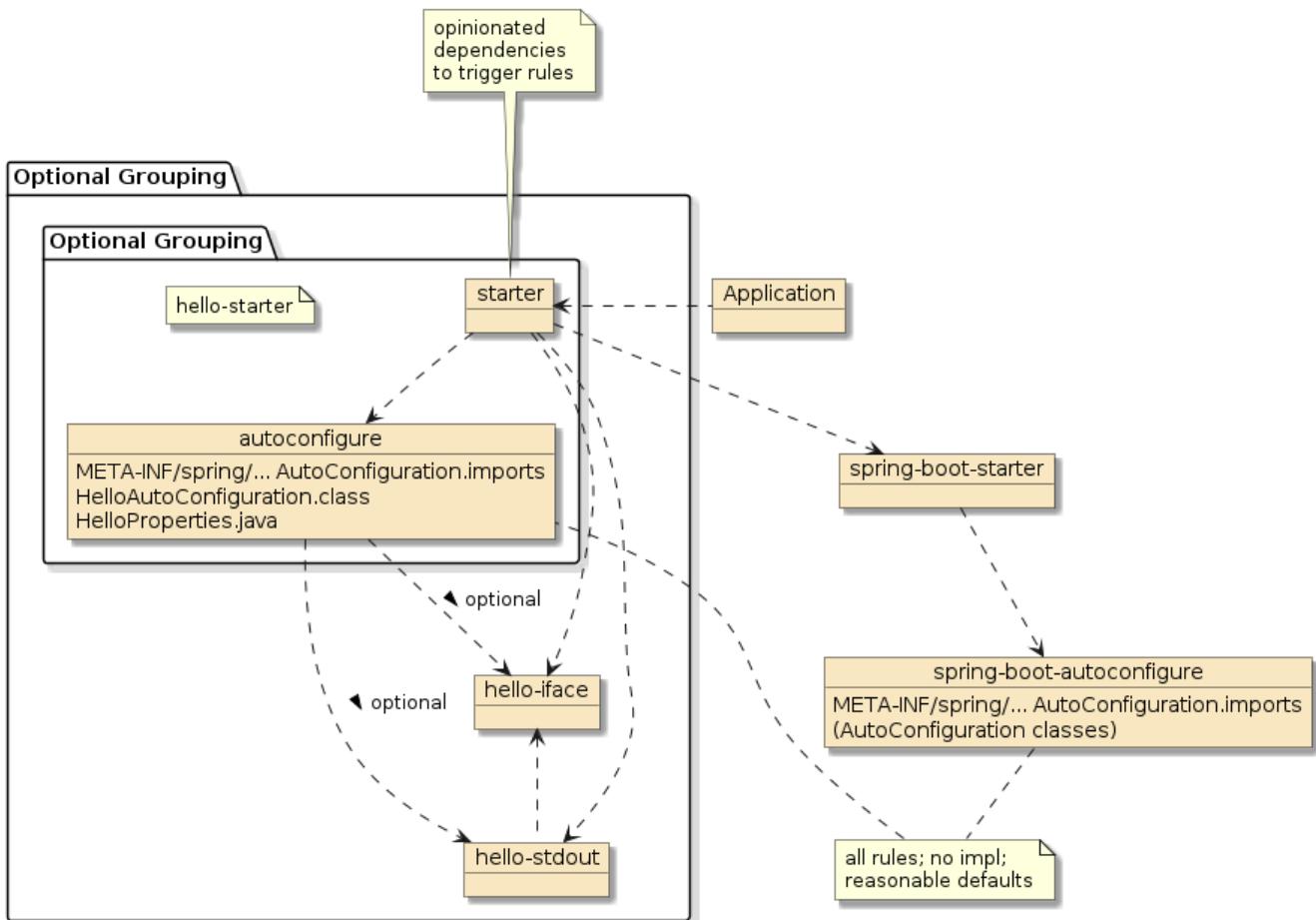
Example Auto-Configuration Module Structure

```
pom.xml
src
`-- main
    |-- java
    |   |-- info
    |   |   |-- ejava
    |   |   |   |-- examples
    |   |   |   |-- app
    |   |   |       |-- hello
    |   |   |           |-- HelloAutoConfiguration.java
    |   |   |           |-- HelloProperties.java
    |-- resources
        |-- META-INF
            |-- spring
                |-- org.springframework.boot.autoconfigure.AutoConfiguration.imports
```

79.7. Auto-Configuration / Starter Roles/Relationships

Modules designed as starters can have varying designs with the following roles carried out:

- Auto-configuration classes that conditionally wire the application. These lay dormant when no conditions to trigger them at startup. They typically lack implementation code, sticking to choices, configuration, and reasonable defaults.
- An opinionated starter with dependencies that trigger the Auto-configuration rules



79.8. Example Starter Module pom.xml

The module commonly termed a **starter** and will have dependencies on

- **spring-boot-starter**, which has dependency on **spring-boot-autoconfigure** with **auto-configuration class** references waiting for conditions
- the service interface
- one or more service implementation(s) and their implementation dependencies

Example Auto-Configuration pom.xml Snippet

```

<groupId>info.ejava.examples.app</groupId>
<artifactId>hello-starter</artifactId>

<dependencies>
    <dependency> ①
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
    </dependency>
    <!-- commonly declares dependency on interface module -->
    <dependency> ②
        <groupId>${project.groupId}</groupId>
        <artifactId>hello-service-api</artifactId>
        <version>${project.version}</version>
    </dependency> ②

```

```

<!-- hello implementation dependency -->
<dependency>
    <groupId>${project.groupId}</groupId>
    <artifactId>hello-service-stdout</artifactId>
    <version>${project.version}</version>
</dependency>

```

- ① dependency on `spring-boot-starter` define classes pertinent to Auto-configuration
- ② `starter` modules commonly define dependencies on interface and implementation modules

The rest of the dependencies have nothing specific to do with Auto-configuration or starter modules, and are there to support the module implementation.

79.9. Application Starter Dependency

The application module declares dependency on the starter module containing or having a dependency on the Auto-configuration artifacts.

Application Module Dependency on Starter Module

```

<!-- takes care of initializing Hello Service for us to inject -->
<dependency>
    <groupId>${project.groupId}</groupId> ①
    <artifactId>hello-starter</artifactId>
    <version>${project.version}</version> ①
</dependency>

```

- ① For this example, the application and starter modules share the same `groupId` and `version` and leverage a `${project}` variable to simplify the expression. That will likely not be the case with most starter module dependencies and will need to be spelled out.

79.10. Starter Brings in Pertinent Dependencies

The starter dependency brings in the Hello Service interface, targeted implementation(s), and some implementation dependencies.

Application Module Transitive Dependencies from Starter

```

$ mvn dependency:tree
...
[INFO] +- info.ejava.examples.app:hello-starter:jar:6.1.0-SNAPSHOT:compile
[INFO] |  +- info.ejava.examples.app:hello-service-api:jar:6.1.0-SNAPSHOT:compile
[INFO] |  +- info.ejava.examples.app:hello-service-stdout:jar:6.1.0-SNAPSHOT:compile
[INFO] |  +- org.projectlombok:lombok:jar:1.18.10:provided
[INFO] |  \- org.springframework.boot:spring-boot-starter-validation:jar:3.3.2:compile
...

```

Chapter 80. Configured Application

The example application contains a component that requests the greeter implementation to say hello to "World".

Injection Point for Auto-configuration Bean

```
import lombok.RequiredArgsConstructor;
...
@Component
@RequiredArgsConstructor ①
public class AppCommand implements CommandLineRunner {
    private final Hello greeter; //<== component in App requires Hello injected

    public void run(String... args) throws Exception {
        greeter.sayHello("World");
    }
}
```

① lombok is being used to provide the constructor injection

80.1. Review: Unconditional Auto-Configuration Class

This starter dependency is bringing in a `@Bean` factory to construct an implementation of `Hello`, that can satisfy the injection dependency.

Example Auto-Configuration Class

```
package info.ejava.examples.app.hello;
...

@Configuration(proxyBeanMethods = false)
@EnableConfigurationProperties(HelloProperties.class)
public class HelloAutoConfiguration {
    @Bean
    public Hello hello(HelloProperties helloProperties) { ①
        return new StdOutHello(helloProperties.getGreeting());
    }
}
```

① Example Auto-configuration configured by `HelloProperties`

This bean will be unconditionally instantiated the way it is currently defined.

80.2. Review: Starter Module Default

The starter dependency brings in an Auto-configuration class that instantiates a `StdOutHello` implementation configured by a `HelloProperties` class.

```
@ConfigurationProperties("hello")
@Data
@Validated
public class HelloProperties {
    @NotBlank
    private String greeting = "HelloProperties default greeting says Hola!"; ①
}
```

① `hello.greeting` default defined in `@ConfigurationProperties` class of starter/autoconfigure module

80.3. Produced Default Starter Greeting

This produces the default greeting

Example Application Execution without Satisfying Property Condition

```
$ java -jar target/appconfig-autoconfig-*-.SNAPSHOT-bootexec.jar
...
HelloProperties default greeting says Hola! World
```

The `HelloAutoConfiguration.hello @Bean` was instantiated with the `HelloProperties` greeting of "HelloProperties default greeting says Hola!". This `Hello` instance was injected into the `AppCommand`, which added "World" to the result.

Example of Reasonable Default



This is an example of a component being Auto-configured with a reasonable default. It did not simply crash, demanding a greeting be supplied.

80.4. User-Application Supplies Property Details

Since the Auto-configuration class is using a properties class, we can define properties (aka "the details") in the main application for the dependency module to use.

`application.properties`

```
#appconfig-autoconfig-example application.properties
#uncomment to use this greeting
hello.greeting: application.properties Says - Hey
```

Runtime Output with `hello.greeting` Property Defined

```
$ java -jar target/appconfig-autoconfig-*-.SNAPSHOT-bootexec.jar
...
application.properties Says - Hey World ①
```

① auto-configured implementation using user-defined property

The same scenario as before is occurring except this time the instantiation of the `HelloProperties` finds a `hello.greeting` property to override the Java default.



Example of Configuring Details

This is an example of customizing the behavior of an Auto-configured component.

Chapter 81. Auto-Configuration Conflict

81.1. Review: Conditional @Bean Factory

We saw how we could make a `@Bean` factory in the User-defined application module conditional (on the value of a property).

Conditional @Bean Factory

```
@SpringBootApplication
public class StarterConfiguredApp {
    ...
    @Bean
    @ConditionalOnProperty(prefix = "hello", name = "quiet", havingValue = "true")
    public Hello quietHello() {
        return new StdOutHello("(hello.quiet property condition set, Application @Bean
says hi)");
    }
}
```

81.2. Potential Conflict

We also saw how to define a `@Bean` factory in an Auto-configuration class brought in by starter module. We now have a condition where the two can cause an ambiguity error that we need to account for.

Example Output with Bean Factory Ambiguity

```
$ java -jar target/appconfig-autoconfig-*-*-SNAPSHOT-bootexec.jar --hello.quiet=true ①
...
*****
APPLICATION FAILED TO START
*****
Description:

Parameter 0 of constructor in info.ejava.examples.app.config.auto.AppCommand
required a single bean, but 2 were found:
    - quietHello: defined by method 'quietHello' in
info.ejava.examples.app.config.auto.StarterConfiguredApp
    - hello: defined by method 'hello' in class path resource
[info/ejava/examples/app/hello/HelloAutoConfiguration.class]
```

This may be due to missing parameter name information

Action:

Consider marking one of the beans as `@Primary`, updating the consumer to accept

multiple beans, or using `@Qualifier` to identify the bean that should be consumed

- ① Supplying the `hello.quiet=true` property value causes two `@Bean` factories to choose from

81.3. `@ConditionalOnMissingBean`

One way to solve the ambiguity is by using the `@ConditionalOnMissingBean` annotation—which defines a condition based on the absence of a bean. Most conditional annotations can be used in both the application and autoconfigure modules. However, the `@ConditionalOnMissingBean` and its sibling `@ConditionalOnBean` are special and meant to be used with Auto-configuration classes in the autoconfigure modules.

Since the Auto-configuration classes are processed after the User-defined classes—there is a clear point to determine whether a User-defined bean does or does not exist. Any other use of these two annotations requires careful ordering and is not recommended.

@ConditionOnMissingBean Auto-Configuration Example

```
...
import org.springframework.boot.autoconfigure.condition.ConditionalOnMissingBean;

@Configuration(proxyBeanMethods = false)
@EnableConfigurationProperties(HelloProperties.class)
public class HelloAutoConfiguration {
    @Bean
    @ConditionalOnMissingBean ①
    public Hello hello(HelloProperties helloProperties) {
        return new StdOutHello(helloProperties.getGreeting());
    }
}
```

- ① `@ConditionOnMissingBean` causes Auto-configured `@Bean` method to be inactive when `Hello` bean already exists

81.4. Bean Conditional Example Output

With the `@ConditionalOnMissingBean` defined on the Auto-configuration class and the property condition satisfied, we get the bean injected from the User-defined `@Bean` factory.

Runtime with Property Condition Satisfied

```
$ java -jar target/appconfig-autoconfig-*-.SNAPSHOT-bootexec.jar --hello.quiet=true
...
(hello.quiet property condition set, Application @Bean says hi) World
```

With the property condition not satisfied, we get the bean injected from the Auto-configuration `@Bean` factory. Wahoo!

Runtime with Property Condition Not Satisfied

```
$ java -jar target/appconfig-autoconfig-*-SNAPSHOT-bootexec.jar  
...  
application.properties Says - Hey World
```

Chapter 82. Resource Conditional and Ordering

We can also define a condition based on the presence of a resource on the filesystem or classpath using the `@ConditionOnResource`. The following example satisfies the condition if the file `hello.properties` exists in the current directory. We are also going to order our Auto-configured classes with the help of the `@AutoConfigureBefore` annotation. There is a sibling `@AutoConfigureAfter` annotation as well as a `AutoConfigureOrder` we could have used.

Example Condition on File Present and Evaluation Ordering

```
...
import org.springframework.boot.autoconfigure.AutoConfigureBefore;
import org.springframework.boot.autoconfigure.condition.ConditionOnResource;

@Configuration(proxyBeanMethods = false)
@ConditionalOnResource(resources = "file:./hello.properties") ①
@AutoConfigureBefore(HelloAutoConfiguration.class) ②
public class HelloResourceAutoConfiguration {
    @Bean
    public Hello resourceHello() {
        return new StdOutHello("hello.properties exists says hello");
    }
}
```

① Auto-configured class satisfied only when file `hello.properties` present

② This Auto-configuration class is processed prior to `HelloAutoConfiguration`

82.1. `@AutoConfiguration` Alternative

We can use the `@AutoConfiguration` annotation, which wraps some of our desired settings.

Example `@AutoConfiguration` Use

```
import org.springframework.boot.autoconfigure.AutoConfiguration;

@AutoConfiguration(before = HelloAutoConfiguration.class)
//==> wraps @Configuration(proxyBeanMethods = false)
//==> wraps @AutoConfigureBefore(HelloAutoConfiguration.class)
@ConditionalOnClass(StdOutHello.class)
@ConditionalOnResource(resources = "file:./hello.properties")
public class HelloResourceAutoConfiguration {
```

82.2. Registering Second Auto-Configuration Class

This second Auto-configuration class is being provided in the same, `hello-starter` module, so we

need to update the `... AutoConfiguration.imports` file. We do this by listing the second class within the same file.

hello-starter AutoConfiguration.imports

```
#  
src/main/resources/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imp  
orts  
  
info.ejava.examples.app.hello.HelloAutoConfiguration  
info.ejava.examples.app.hello.HelloResourceAutoConfiguration
```

82.3. Resource Conditional Example Output

The following execution with `hello.properties` present in the current directory satisfies the condition, causes the `@Bean` factory from `HelloAutoConfiguration` to be skipped because the bean already exists.

Resource Condition Satisfied

```
$ touch hello.properties  
  
$ java -jar target/appconfig-autoconfig-*-SNAPSHOT-bootexec.jar  
...  
hello.properties exists says hello World
```

- when property file is not present:
 - `@Bean` factory from `HelloAutoConfiguration` used since neither property nor resource-based conditions satisfied

Resource Condition Not Satisfied

```
$ rm hello.properties  
$ java -jar target/appconfig-autoconfig-*-SNAPSHOT-bootexec.jar  
...  
application.properties Says - Hey World
```

Chapter 83. @Primary

In the previous example, I purposely put ourselves in a familiar situation to demonstrate an alternative solution if appropriate. We will re-enter the ambiguous match state if we supply a `hello.properties` file and the `hello.quiet=true` property value.

Example Ambiguous Conditional Match

```
$ touch hello.properties
$ java -jar target/appconfig-autoconfig-*-.SNAPSHOT-bootexec.jar --hello.quiet=true
...
*****
APPLICATION FAILED TO START
*****
```

Description:

Parameter 0 of constructor in `info.ejava.examples.app.config.auto.AppCommand` required a single bean,
but 2 were found:
- `quietHello`: defined by method '`quietHello`' in
`info.ejava.examples.app.config.auto.StarterConfiguredApp`
- `resourceHello`: defined by method '`resourceHello`' in class path resource
[`info/ejava/examples/app/hello/HelloResourceAutoConfiguration.class`]

Action:

Consider marking one of the beans as `@Primary`, updating the consumer to accept multiple beans,
or using `@Qualifier` to identify the bean that should be consumed

This time—to correct—we want the resource-based `@Bean` factory to take priority so we add the `@Primary` annotation to our highest priority `@Bean` factory. If there is a conflict—this one will be used.

```
...
import org.springframework.context.annotation.Primary;

@AutoConfiguration(before = HelloAutoConfiguration.class)
@ConditionalOnResource(resources = "file:./hello.properties")
public class HelloResourceAutoConfiguration {
    @Bean
    @Primary //chosen when there is a conflict
    public Hello resourceHello() {
        return new StdOutHello("hello.properties exists says hello");
    }
}
```

83.1. @Primary Example Output

This time we avoid the error with the same conditions met and one of the `@Bean` factories listed as `@Primary` to resolve the conflict.

Ambiguous Choice Resolved through @Primary

```
$ touch hello.properties  
  
$ java -jar target/appconfig-autoconfig-*-SNAPSHOT-bootexec.jar --hello.quiet=true ①  
...  
hello.properties exists says hello World
```

① `@Primary` condition satisfied overrides application `@Bean` condition

Chapter 84. Class Conditions

There are many conditions we can add to our `@Configuration` class or methods. However, there is an important difference between the two.

- class conditional annotations prevent the entire class from loading when not satisfied
- `@Bean` factory conditional annotations allow the class to load but prevent the method from being called when not satisfied

This works for missing classes too! Spring Boot parses the conditional class using `ASM` to detect and then evaluate conditions before allowing the class to be loaded into the JVM. Otherwise, we would get a `ClassNotFoundException` for the import of a class we are trying to base our condition on.

84.1. Class Conditional Example

In the following example, I am adding `@ConditionalOnClass` annotation to prevent the class from being loaded if the implementation class does not exist on the classpath.

```
...
import info.ejava.examples.app.hello.stdout.StdOutHello; ②
import org.springframework.boot.autoconfigure.condition.ConditionalOnClass;

@Configuration(proxyBeanMethods = false)
@ConditionalOnClass(StdOutHello.class) ②
@EnableConfigurationProperties(HelloProperties.class)
public class HelloAutoConfiguration {
    @Bean
    @ConditionalOnMissingBean
    public Hello hello(HelloProperties helloProperties) {
        return new StdOutHello(helloProperties.getGreeting()); ①
    }
}
```

① `StdOutHello` is the implementation instantiated by the `@Bean` factory method

② `HelloAutoConfiguration.class` will not get loaded if `StdOutHello.class` does not exist

The `@ConditionOnClass` accepts either a class or string expression of the fully qualified classname. The sibling `@ConditionalOnMissingClass` accepts only the string form of the classname.



Spring Boot Autoconfigure module contains many examples of real Auto-configuration classes

Chapter 85. Excluding Auto Configurations

We can turn off certain Auto-configured classes using the

- `exclude` attribute of the `@EnableAutoConfiguration` annotation
- `exclude` attribute of the `@SpringBootApplication` annotation which wraps the `@EnableAutoConfiguration` annotation

```
@SpringBootApplication(exclude = {})
// ==> wraps @EnableAutoConfiguration(exclude={})
public class StarterConfiguredApp {
    ...
}
```

Chapter 86. Debugging Auto Configurations

With all these conditional User-defined and Auto-configurations going on, it is easy to get lost or make a mistake. There are two primary tools that can be used to expose the details of the conditional configuration decisions.

86.1. Conditions Evaluation Report

It is easy to get a simplistic textual report of positive and negative condition evaluation matches by adding a `debug` property to the configuration. This can be done by adding `--debug` or `-Ddebug` to the command line.

The following output shows only the positive and negative matching conditions relevant to our example. There is plenty more in the full output.

86.2. Conditions Evaluation Report Example

Conditions Evaluation Report Snippet

```
$ java -jar target/appconfig-autoconfig-*-SNAPSHOT-bootexec.jar --debug | less
...
=====
CONDITIONS EVALUATION REPORT
=====

Positive matches: ①
-----
HelloAutoConfiguration matched:
  - @ConditionalOnClass found required class
'info.ejava.examples.app.hello.stdout.StdOutHello' (OnClassCondition)

HelloAutoConfiguration#hello matched:
  - @ConditionalOnMissingBean (types: info.ejava.examples.app.hello.Hello;
SearchStrategy: all) did not find any beans (OnBeanCondition)

Negative matches: ②
-----
HelloResourceAutoConfiguration:
  Did not match:
    - @ConditionalOnResource did not find resource 'file:./hello.properties'
(OnResourceCondition)
  Matched:
    - @ConditionalOnClass found required class
'info.ejava.examples.app.hello.stdout.StdOutHello' (OnClassCondition)

StarterConfiguredApp#quietHello:
  Did not match:
    - @ConditionalOnProperty (hello.quiet=true) did not find property 'quiet'
```

- ① Positive matches show which conditionals are activated and why
- ② Negative matches show which conditionals are not activated and why

86.3. Condition Evaluation Report Results

The report shows us that

- `HelloAutoConfiguration` class was enabled because `StdOutHello` class was present
- `hello @Bean` factory method of `HelloAutoConfiguration` class was enabled because no other beans were located
- entire `HelloResourceAutoConfiguration` class was not loaded because file `hello.properties` was not present
- `quietHello @Bean` factory method of application class was not activated because `hello.quiet` property was not found

86.4. Actuator Conditions

We can also get a look at the conditionals while the application is running for Web applications using the Spring Boot Actuator. However, doing so requires that we transition our application from a command to a Web application. Luckily this can be done technically by simply changing our starter in the pom.xml file.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
<!--
    <artifactId>spring-boot-starter</artifactId>-->
</dependency>
```

We also need to add a dependency on the `spring-boot-starter-actuator` module.

```
<!-- added to inspect env -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

86.5. Activating Actuator Conditions

The Actuator, by default, will not expose any information without being configured to do so. We can show a JSON version of the Conditions Evaluation Report by adding the `management.endpoints.web.exposure.include` equal to the value `conditions`. I will do that on the command line here. Normally it would be in a profile-specific properties file appropriate for

exposing this information.

Enable Actuator Conditions Report to be Exposed

```
$ java -jar target/appconfig-autoconfig-*-SNAPSHOT-bootexec.jar \
--management.endpoints.web.exposure.include=conditions
```

The Conditions Evaluation Report is available at the following URL: <http://localhost:8080/actuator/conditions>.

Example Actuator Conditions Report

```
{
  "contexts": {
    "application": {
      "positiveMatches": {
        "HelloAutoConfiguration": [
          {
            "condition": "OnClassCondition",
            "message": "@ConditionalOnClass found required class 'info.ejava.examples.app.hello.stdout.StdOutHello'"
          }
        ],
        "HelloAutoConfiguration#hello": [
          {
            "condition": "OnBeanCondition",
            "message": "@ConditionalOnBean (types: info.ejava.examples.app.hello.Hello; SearchStrategy: all) did not find any beans"
          }
        ],
        ...
      },
      "negativeMatches": {
        "StarterConfiguredApp#quietHello": {
          "notMatched": [
            {
              "condition": "OnPropertyCondition",
              "message": "@ConditionalOnProperty (hello.quiet=true) did not find property 'quiet'"
            }
          ],
          "matched": []
        },
        "HelloResourceAutoConfiguration": {
          "notMatched": [
            {
              "condition": "OnResourceCondition",
              "message": "@ConditionalOnResource did not find resource 'file:./hello.properties'"
            }
          ],
          "matched": []
        },
        ...
      }
    }
  }
}
```

86.6. Actuator Environment

It can also be helpful to inspect the environment to determine the value of properties and which source of properties is being used. To see that information, we add `env` to the `exposure.include` property.

Enable Actuator Conditions Report and Environment to be Exposed

```
$ java -jar target/appconfig-autoconfig-*-.SNAPSHOT-bootexec.jar \
--management.endpoints.web.exposure.include=conditions,env
```

86.7. Actuator Links

This adds a full `/env` endpoint and a view specific `/env/{property}` endpoint to see information for a specific property name. The available Actuator links are available at <http://localhost:8080/actuator>.

Actuator Links

```
{
  _links: {
    self: {
      href: "http://localhost:8080/actuator",
      templated: false
    },
    conditions: {
      href: "http://localhost:8080/actuator/conditions",
      templated: false
    },
    env: {
      href: "http://localhost:8080/actuator/env",
      templated: false
    },
    env-toMatch: {
      href: "http://localhost:8080/actuator/env/{toMatch}",
      templated: true
    }
  }
}
```

86.8. Actuator Environment Report

The Actuator Environment Report is available at <http://localhost:8080/actuator/env>.

Example Actuator Environment Report

```
{
  activeProfiles: [ ],
  propertySources: [ {
```

```

        name: "server.ports",
        properties: {
            local.server.port: {
                value: 8080
            }
        }
    },
{
    name: "commandLineArgs",
    properties: {
        management.endpoints.web.exposure.include: {
            value: "conditions,env"
        }
    }
},
...

```

86.9. Actuator Specific Property Source

The source of a specific property and its defined value is available below the `/actuator/env` URI such that the `hello.greeting` property is located at <http://localhost:8080/actuator/env/hello.greeting>.

Example Actuator Environment Report for Specific Property

```

{
    property: {
        source: "applicationConfig: [classpath:/application.properties]",
        value: "application.properties Says - Hey"
    },
}
...
```

86.10. More Actuator

We can explore some of the other Actuator endpoints by changing the include property to `*` and revisiting the main actuator endpoint. [Actuator Documentation](#) is available on the web.

Expose All Actuator Endpoints

```
$ java -jar target/appconfig-autoconfig-*-SNAPSHOT-bootexec.jar \
--management.endpoints.web.exposure.include="*" ①
```

① double quotes ("") being used to escape `*` special character on command line

Chapter 87. Summary

In this module we:

- Defined conditions for `@Configuration` classes and `@Bean` factory methods that are evaluated at runtime startup
- Placed User-defined conditions, which are evaluated first, in with application module
- Placed Auto-configuration classes in separate `starter` module to automatically bootstrap applications with specific capabilities
- Added conflict resolution and ordering to conditions to avoid ambiguous matches
- Discovered how class conditions can help prevent entire `@Configuration` classes from being loaded and disrupt the application because an optional class is missing
- Learned how to debug conditions and visualize the runtime environment through use of the `debug` property or by using the Actuator for web applications

Logging

copyright © 2024 jim stafford (jim.stafford@jhu.edu)

Chapter 88. Introduction

88.1. Why log?

Logging has many uses within an application — spanning:

- auditing actions
- reporting errors
- providing debug information to assist in locating a problem

With much of our code located in libraries — logging is not just for our application code. We will want to know audit, error, and debug information in our library calls as well:

- did that timer fire?
- which calls failed?
- what HTTP headers were input or returned from a REST call?

88.2. Why use a Logger over System.out?

Use of Loggers allow statements to exist within the code that will either:

- be disabled
- log output uninhibited
- log output with additional properties (e.g., timestamp, thread, caller, etc.)

Logs commonly are written to the console and/or files by default — but that is not always the case. Logs can also be exported into centralized servers or database(s) so they can form an integrated picture of a distributed system and provide search and alarm capabilities.



However simple or robust your end logs become, logging starts with the code and is a very important thing to include from the beginning (even if we waited a few modules to cover it).

88.3. Goals

The student will learn:

- to understand the value in using logging over simple System.out.println calls
- to understand the interface and implementation separation of a modern logging framework
- the relationship between the different logger interfaces and implementations
- to use log levels and verbosity to properly monitor the application under different circumstances

- to express valuable context information in logged messages
- to manage logging verbosity
- to configure the output of logs to provide useful information

88.4. Objectives

At the conclusion of this lecture and related exercises, the student will be able to:

1. obtain access to an SLF4J Logger
2. issue log events at different severity levels
3. filter log events based on source and severity thresholds
4. efficiently bypass log statements that do not meet criteria
5. format log events for regular and exception parameters
6. customize log patterns
7. customize appenders
8. add contextual information to log events using Mapped Diagnostic Context
9. trigger additional logging events using Markers
10. use Spring Profiles to conditionally configure logging

Chapter 89. Starting References

There are many resources on the Internet that cover logging, the individual logging implementations, and the Spring Boot opinionated support for logging. You may want to keep a browser window open to one or more of the following starting links while we cover this material. You will not need to go thru all of them, but know there is a starting point to where detailed examples and explanations can be found if not covered in this lesson.

1. [Spring Boot Logging Feature](#) provides documentation from a top-down perspective of how it supplies a common logging abstraction over potentially different logging implementations.
2. [SLF4J Web Site](#) provides documentation, articles, and presentations on SLF4J—the chosen logging interface for Spring Boot and much of the Java community.
3. [Logback Web Site](#) provides a wealth of documentation, articles, and presentations on Logback—the default logging implementation for Spring Boot.
4. [Log4J2 Web Site](#) provides core documentation on Log4J2—a directly supported Spring Boot alternative logging implementation.
5. [Java Util Logging \(JUL\) Documentation Web Site](#) provides an overview of JUL—a lesser supported Spring Boot alternative implementation for logging.

Chapter 90. Logging Dependencies

Most of what we need to perform logging is supplied through our dependency on the [spring-boot-starter](#) and its dependency on [spring-boot-starter-logging](#). The only time we need to supply additional dependencies is when we want to change the default logging implementation or make use of optional, specialized extensions provided by that logging implementation.

Take a look at the transitive dependencies brought in by a straight forward dependency on [spring-boot-starter](#).

Spring Boot Starter Logging Dependencies

```
$ mvn dependency:tree
...
[INFO] info.ejava.examples.app:appconfig-logging-example:jar:6.1.0-SNAPSHOT
[INFO] \- org.springframework.boot:spring-boot-starter:jar:3.3.2:compile
...
[INFO]     +- org.springframework.boot:spring-boot-starter-logging:jar:3.3.2:compile ①
[INFO]         |  +- ch.qos.logback:logback-classic:jar:1.5.6:compile
[INFO]         |  |  +- ch.qos.logback:logback-core:jar:1.5.6:compile
[INFO]         |  |  \- org.slf4j:slf4j-api:jar:2.0.13:compile
[INFO]         |  +- org.apache.logging.log4j:log4j-to-slf4j:jar:2.23.1:compile
[INFO]         |  |  \- org.apache.logging.log4j:log4j-api:jar:2.23.1:compile
[INFO]         |  \- org.slf4j:jul-to-slf4j:jar:2.0.13:compile
...
[INFO]     +- org.springframework:spring-core:jar:3.3.2:compile
[INFO]         |  \- org.springframework:spring-jcl:jar:3.3.2:compile
```

① dependency on [spring-boot-starter](#) brings in [spring-boot-starter-logging](#)

90.1. Logging Libraries

Notice that:

- [spring-core](#) dependency brings in its own repackaging and optimizations of Commons Logging within [spring-jcl](#)
 - [spring-jcl](#) provides a [thin wrapper](#) that looks for logging APIs and self-bootstraps itself to use them—with a preference for the [SLF4J](#) interface, then [Log4J2](#) directly, and then [JUL](#) as a fallback
 - [spring-jcl](#) looks to have replaced the need for [jcl-over-slf4j](#)
- [spring-boot-starter-logging](#) provides dependencies for the [SLF4J API](#), adapters and three optional implementations
 - implementations — these will perform the work behind the SLF4J interface calls
 - [Logback](#) (the default)
 - [Log4J2](#)
 - [Java Util Logging](#)

- adapters — these will bridge the SLF4J calls to the implementations
 - `Logback` implements SLF4J natively - no adapter necessary
 - `log4j-to-slf4j` bridges Log4j to SLF4J
 - `jul-to-slf4j` - bridges Java Util Logging (JUL) to SLF4J



If we use Spring Boot with `spring-boot-starter` right out of the box, we will be using the SLF4J API and Logback implementation configured to work correctly for most cases.

90.2. Spring and Spring Boot Internal Logging

Spring and Spring Boot use an internal version of the [Apache Commons Logging API](#) (Git Repo) (that was previously known as the Jakarta Commons Logging or JCL ([Ref: Wikipedia](#), [Apache Commons Logging](#))) that is rehosted within the `spring-jcl` module to serve as a bridge to different logging implementations (Ref: [Spring Boot Logging](#)).

Chapter 91. Getting Started

OK. We get the libraries we need to implement logging right out of the box with the basic `spring-boot-starter`. How do we get started generating log messages? Lets begin with a comparison with `System.out` so we can see how they are similar and different.

91.1. System.out

`System.out` was built into Java from day 1

- no extra imports are required
- no extra libraries are required

`System.out` writes to wherever `System.out` references. The default is `stdout`. You have seen many earlier examples just like the following.

Example System.out Call

```
@Component
@Profile("system-out") ①
public class SystemOutCommand implements CommandLineRunner {
    public void run(String... args) throws Exception {
        System.out.println("System.out message");
    }
}
```

① restricting component to profile to allow us to turn off unwanted output after this demo

91.2. System.out Output

The example `SystemOutCommand` component above outputs the following statement when called with the `system-out` profile active (using `spring.profiles.active` property).

Example System.out Output

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=system-out ①

System.out message ②
```

① activating profile that turns on our component and turns off all logging

② `System.out` is not impacted by logging configuration and printed to `stdout`

91.3. Turning Off Spring Boot Logging

Where did all the built-in logging (e.g., Spring Boot banner, startup messages, etc.) go in the last example?

The `system-out` profile specified a `logging.level.root` property that effectively turned off all logging.

application-system-out.properties

```
spring.main.banner-mode=off ①  
logging.level.root=OFF ②
```

① turns off printing of verbose Spring Boot startup banner

② turns off all logging (inheriting from the root configuration)



Technically the logging was only turned off for loggers inheriting the root configuration—but we will ignore that detail for right now and just say "all logging".

91.4. Getting a Logger

Logging frameworks make use of the fundamental design idiom—separate interface from implementation. We want our calling code to have simple access to a simple interface to express information to be logged and the severity of that information. We want the implementation to have limitless capability to produce and manage the logs, but want to only pay for what we likely will use. Logging frameworks allow that to occur and provide primary access thru a logging interface and a means to create an instance of that logger. The following diagram shows the basic stereotype roles played by the factory and logger.

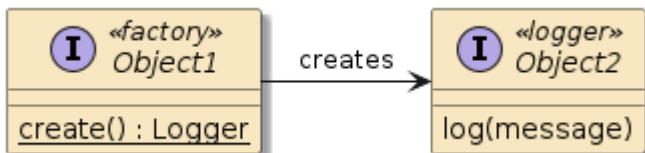


Figure 12. Logging Framework Primary Stereotypes

- Factory creates Logger

Lets take a look at several ways to obtain a Logger using different APIs and techniques.

91.5. Java Util Logger Interface Example

The Java Util Logger (JUL) has been built into Java since 1.4. The primary interface is the `Logger` class. It is used as both the factory and interface for the `Logger` to issue log messages.

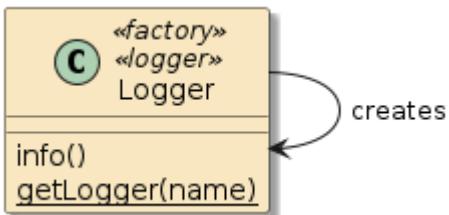


Figure 13. Java Util Logger (JUL) Logger

The following snippet shows an example JUL call.

Example Java Util Logging (JUL) Call

```

package info.ejava.examples.app.config.logging.factory;
...
import java.util.logging.Logger; ①

@Component
public class JULLogger implements CommandLineRunner {
    private static final Logger log = Logger.getLogger(JULLogger.class.getName()); ②

    @Override
    public void run(String... args) throws Exception {
        log.info("Java Util logger message"); ③
    }
}

```

① import the JUL **Logger** class

② get a reference to the JUL **Logger** instance by String name

③ issue a log event



The JUL **Logger** class is used for both the factory and logging interface.

91.6. JUL Example Output

The following output shows that even code using the JUL interface will be integrated into our standard Spring Boot logs.

Example Java Util Logging (JUL) Output

```

java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=factory
...
20:40:54,136 INFO  info.ejava.examples.app.config.logging.factory.JULLogger - Java
Util logger message
...

```



However, JUL is not widely used as an API or implementation. I won't detail it here, but it has been reported to be **much slower** and **missing robust features** of

modern alternatives. That does not mean JUL cannot be used as an API for your code (and the libraries your code relies on) and an implementation for your packaged application. It just means using it as an implementation is uncommon and won't be the default in Spring Boot and other frameworks.

91.7. SLF4J Logger Interface Example

Spring Boot provides first class support for the SLF4J logging interface. The following example shows a sequence similar to the JUL sequence except using `Logger` interface and `LoggerFactory` class from the SLF4J library.

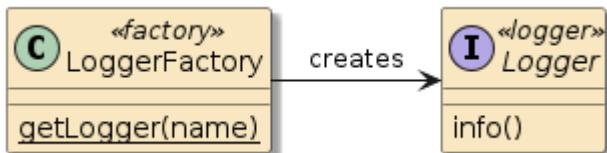


Figure 14. SLF4J `LoggerFactory` and `Logger`

SLF4J Example Call

```
package info.ejava.examples.app.config.logging.factory;

import org.slf4j.Logger; ①
import org.slf4j.LoggerFactory;
...
@Component
public class DeclaredLogger implements CommandLineRunner {
    private static final Logger log = LoggerFactory.getLogger(DeclaredLogger.class);
②

    @Override
    public void run(String... args) throws Exception {
        log.info("declared SLF4J logger message"); ③
    }
}
```

- ① import the SLF4J `Logger` interface and `LoggerFactory` class
- ② get a reference to the SLF4J `Logger` instance using the `LoggerFactory` class
- ③ issue a log event

One immediate improvement SLF4J has over JUL interface is the convenience `getLogger()` method that accepts a class. Loggers are structured in a tree hierarchy and it is common best practice to name them after the fully qualified class that they are called from. The `String` form is also available but the `Class` form helps encourage and simplify following a common best practice.



91.8. SLF4J Example Output

SLF4J Example Output

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=factory ①
...
20:40:55,156 INFO  info.ejava.examples.app.config.logging.factory.DeclaredLogger -
declared SLF4J logger message
...
```

① supplying custom profile to filter output to include only the factory examples

91.9. Lombok SLF4J Declaration Example

Naming loggers after the fully qualified class name is so common that the [Lombok](#) library was able to successfully take advantage of that fact to automate the tasks for adding the imports and declaring the Logger during Java compilation.

Lombok Example Call

```
package info.ejava.examples.app.config.logging.factory;

import lombok.extern.slf4j.Slf4j;
...
@Component
@Slf4j ①
public class LombokDeclaredLogger implements CommandLineRunner {
    ②
        @Override
        public void run(String... args) throws Exception {
            log.info("lombok declared SLF4J logger"); ③
        }
}
```

① `@Slf4j` annotation automates the import statements and Logger declaration

② Lombok will declare a static `log` property using `LoggerFactory` during compilation

③ normal log statement provided by calling class — no different from earlier example

91.10. Lombok Example Output

Since Lombok primarily automates code generation at compile time, the produced output is identical to the previous manual declaration example.

Lombok Example Output

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=factory
```

```
...
20:40:55,155 INFO  info.ejava.examples.app.config.logging.factory.LombokDeclaredLogger
- lombok declared SLF4J logger message
...
```

91.11. Lombok Dependency

Of course, we need to add the following dependency to the project `pom.xml` to enable Lombok annotation processing.

Lombok Dependency

```
<!-- used to declare logger -->
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <scope>provided</scope>
</dependency>
```

Chapter 92. Logging Levels

The `Logger` returned from the `LoggerFactory` will have an associated `level` assigned elsewhere—that represents its verbosity threshold. We issue messages to the `Logger` using separate methods that indicate their severity.

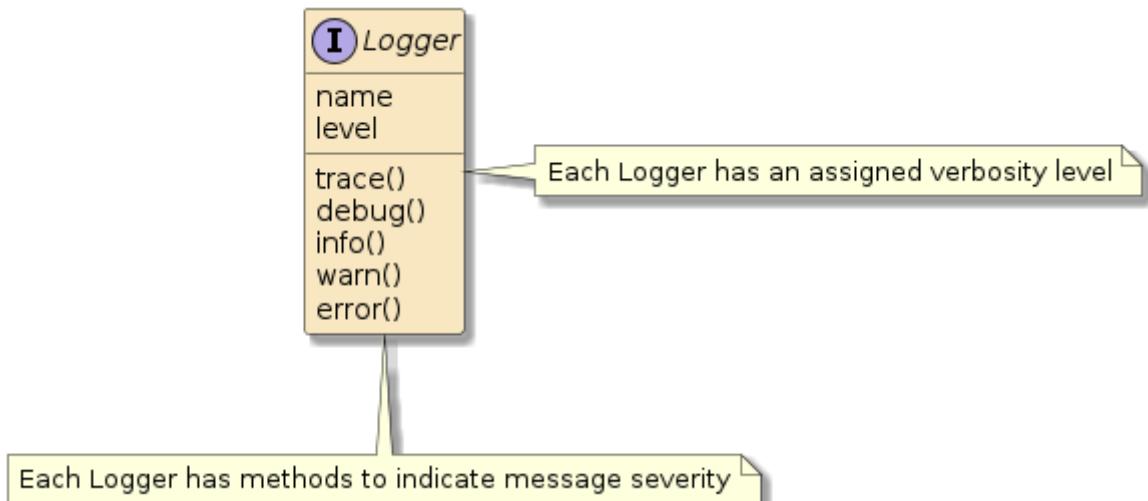


Figure 15. Logging Level

The logging severity level calls supported by SLF4J span from `trace()` to `error()`. The logging threshold levels supported by Spring Boot and Logback also span from `TRACE` to `ERROR` and include an `OFF` (all levels are case insensitive). When we compare levels and thresholds—treat `TRACE` being less severe than `ERROR`.



Severity levels supported by other APIs ([JUL Levels](#), [Log4J2 Levels](#)) are mapped to levels supported by SLF4J.

92.1. Common Level Use

Although there cannot be enforcement of when to use which level—there are common conventions. The following is a set of conventions I live by:

TRACE

Detailed audits events and verbose state of processing

- example: Detailed state at a point in the code. SQL, params, and results of a query.

DEBUG

Audit events and state giving insight of actions performed

- example: Beginning/ending a decision branch or a key return value

INFO

Notable audit events and state giving some indication of overall activity performed

- example: Started/completed transaction for purchase

WARN

Something unusual to highlight but the application was able to recover

- example: Read timeout for remote source

ERROR

Significant or unrecoverable error occurred and an important action failed. These should be extremely limited in their use.

- example: Cannot connect to database

92.2. Log Level Adjustments

Obviously, there are no perfect guidelines. Adjustments need to be made on a case by case basis.



When forming your logging approach — ask yourself "are the logs telling me what I need to know when I look under the hood?", "what can they tell me with different verbosity settings?", and "what will it cost in terms of performance and storage?".



The last thing you want to do is to be called in for a problem and the logs tell you nothing or too much of the wrong information. Even worse — changing the verbosity of the logs will not help for when the issue occurs the next time.

92.3. Logging Level Example Calls

The following is an example of making very simple calls to the logger at different severity levels.

Logging Level Example Calls

```
package info.ejava.examples.app.config.logging.levels;  
...  
@Slf4j  
public class LoggerLevels implements CommandLineRunner {  
    @Override  
    public void run(String... args) throws Exception {  
        log.trace("trace message"); ①  
        log.debug("debug message");  
        log.info("info message");  
        log.warn("warn message");  
        log.error("error message"); ①  
    }  
}
```

① example issues one log message at each of the available LSF4J severity levels

92.4. Logging Level Output: INFO

This example references a simple profile that configures loggers for our package to report at the

`INFO` severity level to simulate the default.

Logging Level INFO Example Output

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=levels ①

06:36:15,910 INFO  info.ejava.examples.app.config.logging.levels.LoggerLevels - info
message ②
06:36:15,912 WARN  info.ejava.examples.app.config.logging.levels.LoggerLevels - warn
message
06:36:15,912 ERROR info.ejava.examples.app.config.logging.levels.LoggerLevels - error
message
```

① profile sets `info.ejava.examples.app.config.logging.levels` threshold level to `INFO`

② messages logged for `INFO`, `WARN`, and `ERROR` severity because they are \geq `INFO`

The referenced profile turns off all logging except for the `info.ejava...levels` package being demonstrated and customizes the pattern of the logs. We will look at that more soon.

application-levels.properties

```
#application-levels.properties
logging.pattern.console=%date{HH:mm:ss.SSS} %-5level %logger - %msg%n ③

logging.level.info.ejava.examples.app.config.logging.levels=info ②
logging.level.root=OFF ①
```

① all loggers are turned off by default

② example package logger threshold level produces log events with severity \geq `INFO`

③ customized console log messages to contain pertinent example info

92.5. Logging Level Output: DEBUG

Using the command line to express a `logging.level` property, we lower the threshold for our logger to `DEBUG` and get one additional severity level in the output.

Logging Level DEBUG Example Output

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=levels \
--logging.level.info.ejava.examples.app.config.logging.levels=DEBUG ①

06:37:04,292 DEBUG info.ejava.examples.app.config.logging.levels.LoggerLevels - debug
message ②
06:37:04,293 INFO  info.ejava.examples.app.config.logging.levels.LoggerLevels - info
message
06:37:04,294 WARN  info.ejava.examples.app.config.logging.levels.LoggerLevels - warn
message
```

```
06:37:04,294 ERROR info.ejava.examples.app.config.logging.levels.LoggerLevels - error message
```

- ① logging.level sets `info.ejava.examples.app.config.logging.levels` threshold level to DEBUG
- ② messages logged for DEBUG, INFO, WARN, and ERROR severity because they are >= DEBUG

92.6. Logging Level Output: TRACE

Using the command line to express a `logging.level` property, we lower the threshold for our logger to `TRACE` and get two additional severity levels in the output over what we produced with `INFO`.

Logging Level TRACE Example Output

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=levels \
--logging.level.info.ejava.examples.app.config.logging.levels=TRACE ①

06:37:19,968 TRACE info.ejava.examples.app.config.logging.levels.LoggerLevels - trace message ②
06:37:19,970 DEBUG info.ejava.examples.app.config.logging.levels.LoggerLevels - debug message
06:37:19,970 INFO  info.ejava.examples.app.config.logging.levels.LoggerLevels - info message
06:37:19,970 WARN  info.ejava.examples.app.config.logging.levels.LoggerLevels - warn message
06:37:19,970 ERROR info.ejava.examples.app.config.logging.levels.LoggerLevels - error message
```

- ① logging.level sets `info.ejava.examples.app.config.logging.levels` threshold level to TRACE
- ② messages logged for all severity levels because they are >= TRACE

92.7. Logging Level Output: WARN

Using the command line to express a `logging.level` property, we raise the threshold for our logger to `WARN` and get one less severity level in the output over what we produced with `INFO`.

Logging Level WARN Example Output

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=levels \
--logging.level.info.ejava.examples.app.config.logging.levels=WARN ①

06:37:32,753 WARN  info.ejava.examples.app.config.logging.levels.LoggerLevels - warn message ②
06:37:32,755 ERROR info.ejava.examples.app.config.logging.levels.LoggerLevels - error message
```

- ① logging.level sets `info.ejava.examples.app.config.logging.levels` threshold level to WARN

- ② messages logged for `WARN`, and `ERROR` severity because they are $\geq \text{WARN}$

92.8. Logging Level Output: OFF

Using the command line to express a `logging.level` property, we set the threshold for our logger to `OFF` and get no output produced.

Logging Level OFF Example Output

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=levels \
--logging.level.info.ejava.examples.app.config.logging.levels=OFF ①
```

②

① `logging.level` sets `info.ejava.examples.app.config.logging.levels` threshold level to `OFF`

② no messages logged because logger turned off

Chapter 93. Discarded Message Expense

The designers of logger frameworks are well aware that excess logging— even statements that are disabled— can increase the execution time of a library call or overall application. We have already seen how severity level thresholds can turn off output and that gives us substantial savings within the logging framework itself. However, we must be aware that building a message to be logged can carry its own expense and be aware of the tools to mitigate the problem.

Assume we have a class that is relatively expensive to obtain a String representation.

Example Expensive `toString()`

```
class ExpensiveToLog {  
    public String toString() { ①  
        try { Thread.sleep(1000); } catch (Exception ignored) {}  
        return "hello";  
    }  
}
```

① calling `toString()` on instances of this class will incur noticeable delay

93.1. Blind String Concatenation

Now lets say we create a message to log through straight, eager String concatenation. What is wrong here?

Blind String Concatenation Example

```
ExpensiveToLog obj=new ExpensiveToLog();  
// ...  
log.debug("debug for expensiveToLog: " + obj + "!");
```

1. The log message will get formed by eagerly concatenating several Strings together
2. One of those Strings is produced by a relatively expensive `toString()` method
3. **Problem:** The work of eagerly forming the String is wasted if `DEBUG` is not enabled

93.2. Verbosity Check

Assuming the information from the `toString()` call is valuable and needed when we have `DEBUG` enabled—a verbosity check is one common solution we can use to determine if the end result is worth the work. There are two very similar ways we can do this.

The first way is to dynamically check the current threshold level of the logger within the code and only execute if the requested severity level is enabled. We are still going to build the relatively expensive String when `DEBUG` is enabled but we are going to save all that processing time when it is not enabled. This overall approach of using a code block works best when creating the message requires multiple lines of code. This specific technique of dynamically checking is suitable when

there are very few checks within a class.

93.2.1. Dynamic Verbosity Check

The first way is to dynamically check the current threshold level of the logger within the code and only execute if the requested severity level is enabled. We are still going to build the relatively expensive String when `DEBUG` is enabled but we are going to save all that processing time when it is not enabled. This overall approach of using a code block works best when creating the message requires multiple lines of code. This specific technique of dynamically checking is suitable when there are very few checks within a class.

Dynamic Verbosity Check Example

```
if (log.isDebugEnabled()) { ①
    log.debug("debug for expensiveToLog: " + obj + "!");
}
```

① code block with expensive `toString()` call is bypassed when `DEBUG` disabled

93.2.2. Static Final Verbosity Check

A variant of the first approach is to define a `static final boolean` variable at the start of the class, equal to the result of the enabled test. This variant allows the JVM to know that the value of the `if` predicate will never change allowing the code block and further checks to be eliminated when disabled. This alternative is better when there are multiple blocks of code that you want to make conditional on the threshold level of the logger. This solution assumes the logger threshold will never be changed or that the JVM will be restarted to use the changed value. I have seen this technique commonly used in `libraries` where they anticipate many calls and they are commonly judged on their method throughput performance.

Static Verbosity Check Example

```
private static final boolean DEBUG_ENABLED = log.isDebugEnabled(); ①
...
if (DEBUG_ENABLED) { ②
    log.debug("debug for expensiveToLog: " + obj + "!");
}
...
```

① logger's verbosity level tested when class loaded and stored in `static final` variable

② code block with expensive `toString()`

93.3. SLF4J Parameterized Logging

SLF4J API offers another solution that removes the need for the `if` clause—thus cleaning your code of those extra conditional blocks. The SLF4J `Logger` interface has a `format` and `args` variant for each verbosity level call that permits the threshold to be consulted prior to converting any of the parameters to a String.

The format specification uses a set of curly braces ("{}") to express an insertion point for an ordered set of arguments. There are no format options. It is strictly a way to lazily call `toString()` on each argument and insert the result.

SLF4J Parameterized Logging Example

```
log.debug("debug for expensiveToLog: {}!", obj); ① ②
```

① {} is a placeholder for the result of `obj.toString()` if called

② `obj.toString()` only called and overall message concatenated if logger threshold set to \leq DEBUG

93.4. Simple Performance Results: Disabled

Not scientific by any means, but the following results try to highlight the major cost differences between blind concatenation and the other methods. The basic results also show the parameterized logging technique to be on par with the threshold level techniques with far less code complexity.

The test code warms up the logger with a few calls and then issues the debug statements shown above in succession with time hacks taken in between each.

The first set of results are from logging threshold set to `INFO`. The blind concatenation shows that it eagerly calls the `obj.toString()` method just to have its resulting message discarded. The other methods do not pay a measurable penalty.

- test code
 - warms up logger with few calls
 - issues the debug statements shown above in succession
 - time hacks taken in between each
- first set of results are from logging threshold set to `INFO`
 - blind concatenation shows it eagerly calls the `obj.toString()` method just to have its resulting message discarded
 - other methods do not pay a measurable penalty

Disabled Logger Relative Results

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=expense \
--logging.level.info.ejava.examples.app.config.logging.expense=INFO
```

```
11:44:25.462 INFO  info.ejava.examples.app.config.logging.expense.DisabledOptimization
- warmup logger
11:44:26.476 INFO  info.ejava.examples.app.config.logging.expense.DisabledOptimization
- \
```

```
concat: 1012, ifDebug=0, DEBUG_ENABLED=0, param=0 ① ②
```

① eager blind concatenation pays `toString()` cost even when not needed (1012ms)

② verbosity check and lazy parameterized logging equally efficient (0ms)

93.5. Simple Performance Results: Enabled

The second set of results are from logging threshold set to `DEBUG`. You can see that causes the relatively expensive `toString()` to be called for each of the four techniques shown with somewhat equal results. I would not put too much weight on a few milliseconds difference between the calls here except to know that neither provide a noticeable processing delay over the other when the logging threshold has been met.

Enabled Logger Relative Results

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=expense \
--logging.level.info.ejava.examples.app.config.logging.expense=DEBUG

11:44:43.560 INFO  info.ejava.examples.app.config.logging.expense.DisabledOptimization
- warmup logger
11:44:43.561 DEBUG info.ejava.examples.app.config.logging.expense.DisabledOptimization
- warmup logger
11:44:44.572 DEBUG info.ejava.examples.app.config.logging.expense.DisabledOptimization
- debug for expensiveToLog: hello!
11:44:45.575 DEBUG info.ejava.examples.app.config.logging.expense.DisabledOptimization
- debug for expensiveToLog: hello!
11:44:46.579 DEBUG info.ejava.examples.app.config.logging.expense.DisabledOptimization
- debug for expensiveToLog: hello!
11:44:46.579 DEBUG info.ejava.examples.app.config.logging.expense.DisabledOptimization
- debug for expensiveToLog: hello!
11:44:47.582 INFO  info.ejava.examples.app.config.logging.expense.DisabledOptimization
- \
concat: 1010, ifDebug=1003, DEBUG_ENABLED=1004, param=1003 ①
```

① all four methods paying the cost of the relatively expensive `obj.toString()` call

Chapter 94. Exception Logging

SLF4J interface and parameterized logging goes one step further to also support `Exceptions`. If you pass an `Exception` object as the last parameter in the list—it is treated special and will not have its `toString()` called with the rest of the parameters. Depending on the configuration in place, the stack trace for the `Exception` is logged instead. The following snippet shows an example of an `Exception` being thrown, caught, and then logged.

Example Exception Logging

```
public void run(String... args) throws Exception {
    try {
        log.info("calling iThrowException");
        iThrowException();
    } catch (Exception ex) {
        log.warn("caught exception", ex); ①
    }
}

private void iThrowException() throws Exception {
    throw new Exception("example exception");
}
```

① `Exception` passed to logger with message

94.1. Exception Example Output

When we run the example, note that the message is printed in its normal location and a stack trace is added for the supplied `Exception` parameter.

Example Exception Logging Output

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=exceptions

13:41:17.119 INFO  info.ejava.examples.app.config.logging.exceptions.ExceptionExample
- calling iThrowException
13:41:17.121 WARN  info.ejava.examples.app.config.logging.exceptions.ExceptionExample
- caught exception ①
java.lang.Exception: example exception ②
    at
info.ejava.examples.app.config.logging.exceptions.ExceptionExample.iThrowException(ExceptionExample.java:23)
    at
info.ejava.examples.app.config.logging.exceptions.ExceptionExample.run(ExceptionExample.java:15)
    at
org.springframework.boot.SpringApplication.callRunner(SpringApplication.java:784)
...
```

```
at org.springframework.boot.loader.Launcher.launch(Launcher.java:51)
at org.springframework.boot.loader.JarLauncher.main(JarLauncher.java:52)
```

- ① normal message logged
- ② stack trace for last `Exception` parameter logged

94.2. Exception Logging and Formatting

Note that you can continue to use parameterized logging with Exceptions. The message passed in above was actually a format with no parameters. The snippet below shows a format with two parameters and an `Exception`.

Example Exception Logging with Parameters

```
log.warn("caught exception {} {}", "p1", "p2", ex);
```

The first two parameters are used in the formatting of the core message. The last `Exception` parameters is printed as a regular exception.

Example Exception Logging with Parameters Output

```
13:41:17.119 INFO  info.ejava.examples.app.config.logging.exceptions.ExceptionExample
- calling iThrowException
13:41:17.122 WARN  info.ejava.examples.app.config.logging.exceptions.ExceptionExample
- caught exception p1 p2 ①
java.lang.Exception: example exception ②
    at
info.ejava.examples.app.config.logging.exceptions.ExceptionExample.iThrowException(ExceptionExample.java:23)
    at
info.ejava.examples.app.config.logging.exceptions.ExceptionExample.run(ExceptionExample.java:15)
    at
org.springframework.boot.SpringApplication.callRunner(SpringApplication.java:784)
...
    at org.springframework.boot.loader.Launcher.launch(Launcher.java:51)
    at org.springframework.boot.loader.JarLauncher.main(JarLauncher.java:52)
```

- ① two early parameters ("p1" and "p2") where used to complete the message template
- ② `Exception` passed as the last parameter had its stack trace logged

Chapter 95. Logging Pattern

Each of the previous examples showed logging output using a particular pattern. The pattern was expressed using a `logging.pattern.console` property. The [Logback Conversion Documentation](#) provides details about how the logging pattern is defined.

Sample Custom Pattern

```
logging.pattern.console=%date{HH:mm:ss.SSS} %-5level %logger - %msg%
```

The pattern consisted of:

- `%date` (or `%d`) - time of day down to millisecs
- `%level` (or `%p`, `%le`) - severity level left justified and padded to 5 characters
- `%logger` (or `%c`, `%lo`) - full name of logger
- `%msg` (or `%m`, `%message`) - full logged message
- `%n` - operating system-specific new line

If you remember, that produced the following output.

Review: LoggerLevels Example Pattern Output

```
java -jar target/appconfig-logging-example-*-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=levels

06:00:38.891 INFO  info.ejava.examples.app.config.logging.levels.LoggerLevels - info
message
06:00:38.892 WARN   info.ejava.examples.app.config.logging.levels.LoggerLevels - warn
message
06:00:38.892 ERROR  info.ejava.examples.app.config.logging.levels.LoggerLevels - error
message
```

95.1. Default Console Pattern

Spring Boot comes out of the box with a slightly more verbose [default pattern](#) expressed with the `CONSOLE_LOG_PATTERN` property. The following snippet depicts the information found within the Logback property definition — with some new lines added in to help read it.

Gist of CONSOLE_LOG_PATTERN from GitHub

```
%clr(%d${${LOG_DATEFORMAT_PATTERN:-yyyy-MM-dd HH:mm:ss.SSS}}){faint}
%clr(${LOG_LEVEL_PATTERN:-%5p})
%clr(${PID:- }){magenta}
%clr(---){faint}
%clr([%15.15t]){faint}
%clr(%-40.40logger{39}){cyan}
%clr(:){faint}
```

```
%m%n${LOG_EXCEPTION_CONVERSION_WORD:-%wEx}
```

You should see some familiar conversion words from my earlier pattern example. However, there are some additional conversion words used as well. Again, keep the [Logback Conversion Documentation](#) close by to lookup any additional details.

- %d - timestamp defaulting to full format
- %p - severity level right justified and padded to 5 characters
- \$PID - system property containing the process ID
- %t (or %thread) - thread name right justified and padded to 15 characters and chopped at 15 characters
- %logger - logger name optimized to fit within 39 characters , left justified and padded to 40 characters, chopped at 40 characters
- %m - fully logged message
- %n - operating system-specific new line
- %wEx - [Spring Boot-defined exception formatting](#)

95.2. Default Console Pattern Output

We will take a look at conditional variable substitution in a moment. This next example reverts to the default [CONSOLE_LOG_PATTERN](#).

LoggerLevels Output with Default Spring Boot Console Log Pattern

```
java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--logging.level.root=OFF \
--logging.level.info.ejava.examples.app.config.logging.levels.LoggerLevels=TRACE

2020-03-27 06:31:21.475 TRACE 31203 --- [           main]
i.e.e.a.c.logging.levels.LoggerLevels : trace message
2020-03-27 06:31:21.477 DEBUG 31203 --- [           main]
i.e.e.a.c.logging.levels.LoggerLevels : debug message
2020-03-27 06:31:21.477 INFO 31203 --- [          main]
i.e.e.a.c.logging.levels.LoggerLevels : info message
2020-03-27 06:31:21.477 WARN 31203 --- [          main]
i.e.e.a.c.logging.levels.LoggerLevels : warn message
2020-03-27 06:31:21.477 ERROR 31203 --- [         main]
i.e.e.a.c.logging.levels.LoggerLevels : error message
```

Spring Boot defines color coding for the console that is not visible in the text of this document. The color for severity level is triggered by the level—red for [ERROR](#), yellow for [WARN](#), and green for the other three levels.

```

2020-03-27 06:31:21.475 TRACE 31203 --- [           main] i.e.e.a.c.logging.levels.LoggerLevels : trace message
2020-03-27 06:31:21.477 DEBUG 31203 --- [           main] i.e.e.a.c.logging.levels.LoggerLevels : debug message
2020-03-27 06:31:21.477 INFO  31203 --- [           main] i.e.e.a.c.logging.levels.LoggerLevels : info message
2020-03-27 06:31:21.477 WARN  31203 --- [           main] i.e.e.a.c.logging.levels.LoggerLevels : warn message
2020-03-27 06:31:21.477 ERROR 31203 --- [           main] i.e.e.a.c.logging.levels.LoggerLevels : error message

```

Figure 16. Default Spring Boot Console Log Pattern Coloring

95.3. Variable Substitution

Logging configurations within Spring Boot make use of variable substitution. The value of `LOG_DATEFORMAT_PATTERN` will be applied wherever the expression `${LOG_DATEFORMAT_PATTERN}` appears. The " `${}`" characters are part of the variable expression and will not be part of the result.

95.4. Conditional Variable Substitution

Variables can be defined with default values in the event they are not defined. In the following expression `${LOG_DATEFORMAT_PATTERN:-yyyy-MM-dd HH:mm:ss.SSS}`:

- the value of `LOG_DATEFORMAT_PATTERN` will be used if defined
- the value of "yyyy-MM-dd HH:mm:ss.SSS" will be used if not defined



The " `${}`" and embedded ":"- characters following the variable name are part of the expression when appearing within an XML configuration file and will not be part of the result. The dash (-) character should be removed if using within a property definition.

95.5. Date Format Pattern

As we saw from a peek at the Spring Boot `CONSOLE_LOG_PATTERN` default definition, we can change the format of the timestamp using the `LOG_DATEFORMAT_PATTERN` system property. That system property can flexibly be set using the `logging.pattern.dateformat` property. See the [Spring Boot Documentation](#) for information on this and other properties. The following example shows setting that property using a command line argument.

Setting Date Format

```

$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--logging.level.root=OFF \
--logging.level.info.ejava.examples.app.config.logging.levels.LoggerLevels=INFO \
--logging.pattern.dateformat="HH:mm:ss.SSS" ①

08:20:42.939 INFO 39013 --- [           main] i.e.e.a.c.logging.levels.LoggerLevels
: info message
08:20:42.942 WARN 39013 --- [           main] i.e.e.a.c.logging.levels.LoggerLevels
: warn message
08:20:42.942 ERROR 39013 --- [          main] i.e.e.a.c.logging.levels.LoggerLevels
: error message

```

① setting `LOG_DATEFORMAT_PATTERN` using `logging.pattern.dateformat` property

95.6. Log Level Pattern

We also saw from the default definition of `CONSOLE_LOG_PATTERN` that the severity level of the output can be changed using the `LOG_LEVEL_PATTERN` system property. That system property can be flexibly set with the `logging.pattern.level` property. The following example shows setting the format to a single character, left justified. Therefore, we can map `INFO`⇒`I`, `WARN`⇒`W`, and `ERROR`⇒`E`.

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--logging.level.root=OFF \
--logging.level.info.ejava.examples.app.config.logging.levels.LoggerLevels=INFO \
--logging.pattern.dateformat="HH:mm:ss.SSS" \
--logging.pattern.level="%.-1p" ①
②
08:59:17.376 I 44756 --- [           main] i.e.e.a.c.logging.levels.LoggerLevels   :
info message
08:59:17.379 W 44756 --- [           main] i.e.e.a.c.logging.levels.LoggerLevels   :
warn message
08:59:17.379 E 44756 --- [           main] i.e.e.a.c.logging.levels.LoggerLevels   :
error message
```

① `logging.level.pattern` expressed to be 1 character, left justified

② single character produced in console log output

95.7. Conversion Pattern Specifiers

[Spring Boot Features Web Page](#) documents some formatting rules. However, more details on the parts within the conversion pattern are located on the [Logback Pattern Layout Web Page](#). The overall end-to-end pattern definition I have shown you is called a "Conversion Pattern". Conversion Patterns are made up of:

- Literal Text (e.g., `---`, whitespace, `:`)—hard-coded strings providing decoration and spacing for conversion specifiers
- Conversion Specifiers - (e.g., `%-40.40logger{39}`)—an expression that will contribute a formatted property of the current logging context
 - starts with `%`
 - followed by format modifiers—(e.g., `-40.40`)—addresses min/max spacing and right/left justification
 - optionally provide minimum number of spaces
 - use a negative number (`-#`) to make it left justified and a positive number (`#`) to make it right justified
 - optionally provide maximum number of spaces using a decimal place and number (`.#`). Extra characters will be cut off
 - use a negative number (`.-#`) to start from the left and positive number (`.#`) to start from the right

- followed by a conversion word (e.g., `logger`, `msg`)—keyword name for the property
- optional parameters (e.g., `{39}`)—see individual conversion words for details on each

95.8. Format Modifier Impact Example

The following example demonstrates how the different format modifier expressions can impact the `level` property.

Table 5. %level Format Modifier Impact Example

<code>logging.pattern.level</code>	<code>output</code>	<code>comment</code>
<code>[%level]</code>	<code>[INFO]</code> <code>[WARN]</code> <code>[ERROR]</code>	value takes whatever space necessary
<code>[%6level]</code>	<code>[INFO]</code> <code>[WARN]</code> <code>[ERROR]</code>	value takes at least 6 characters, right justified
<code>[%-6level]</code>	<code>[INFO]</code> <code>[WARN]</code> <code>[ERROR]</code>	value takes at least 6 characters, left justified
<code>[%.2level]</code>	<code>[IN]</code> <code>[WA]</code> <code>[ER]</code>	value takes no more than 2 characters, starting from the left
<code>[%.2level]</code>	<code>[FO]</code> <code>[RN]</code> <code>[OR]</code>	value takes no more than 2 characters, starting from the right

95.9. Example Override

Earlier you saw how we could control the console pattern for the `%date` and `%level` properties. To go any further, we are going to have to override the entire `CONSOLE_LOG_PATTERN` system property and can define it using the `logging.pattern.console` property.

That is too much to define on the command line, so lets move our definition to a profile-based property file (`application-layout.properties`)

application-layout.properties

```
#application-layout.properties ①

#default to time-of-day for the date
logging.pattern.dateformat=HH:mm:ss.SSS
#supply custom console layout
logging.pattern.console=%clr(%d${${LOG_DATEFORMAT_PATTERN:HH:mm:ss.SSS}}){faint} \
%clr(${LOG_LEVEL_PATTERN:%5p}) \
%clr(-){faint} \
```

```
%clr(%27logger{40}){cyan}\
%clr({faint}\
%clr(%method){cyan} \ ②
%clr(:){faint}\
%clr(%line){cyan} \ ②
%m%n\
${LOG_EXCEPTION_CONVERSION_WORD:%wEx}

logging.level.info.ejava.examples.app.config.logging.levels.LoggerLevels=INFO
logging.level.root=OFF
```

① property file used when `layout` profile active

② customization added `method` and `line` of caller — at a processing expense

95.10. Expensive Conversion Words

I added two new helpful properties that could be considered controversial because they require extra overhead to obtain that information from the JVM. The technique has commonly involved throwing and catching an exception internally to determine the calling location from the self-generated stack trace:

- `%method` (or `%M`) - name of method calling logger
- `%line` (or `%L`) - line number of the file where logger call was made



The additional "expensive" fields are being used for console output for demonstrations using a demonstration profile. Consider your log information needs on a case-by-case basis and learn from this lesson what and how you can modify the logs for your specific needs. For example — to debug an error, you can switch to a more detailed and verbose profile without changing code.

95.11. Example Override Output

We can activate the profile and demonstrate the modified format using the following command.

Example Console Pattern Override Output

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=layout

14:25:58.428 INFO - logging.levels.LoggerLevels#run:14 info message
14:25:58.430 WARN - logging.levels.LoggerLevels#run:15 warn message
14:25:58.430 ERROR - logging.levels.LoggerLevels#run:16 error message
```

The coloring does not show up above so the image below provides a perspective of what that looks like.

```
14:25:58.428 INFO - logging.levels.LoggerLevels#run:14 info message
14:25:58.430 WARN - logging.levels.LoggerLevels#run:15 warn message
14:25:58.430 ERROR - logging.levels.LoggerLevels#run:16 error message
```

Figure 17. Example Console Pattern Override Coloring

95.12. Layout Fields

Please see the [Logback Layouts Documentation](#) for a detailed list of conversion words and how to optionally format them.

Chapter 96. Loggers

We have demonstrated a fair amount capability thus far without having to know much about the internals of the logger framework. However, we need to take a small dive into the logging framework in order to explain some further concepts.

- Logger Ancestry
- Logger Inheritance
- Appenders
- Logger Additivity

96.1. Logger Tree

Loggers are organized in a hierarchy starting with a root logger called "root". As you would expect, higher in the tree are considered ancestors and lower in the tree are called descendants.

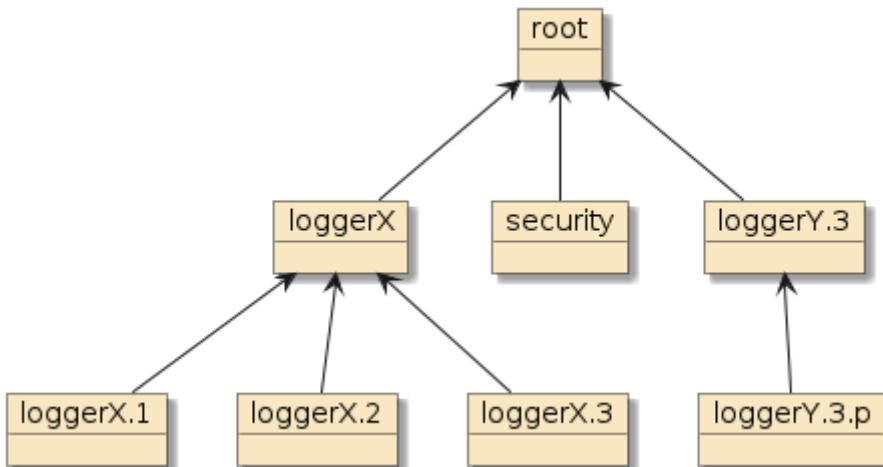


Figure 18. Example Logger Tree

Except for root, the ancestor/descendant structure of loggers depends on the hierarchical name of each logger. Based on the loggers in the diagram

- X, Y.3, and security are descendants and direct children of root
- Y.3 is example of logger lacking an explicitly defined parent in hierarchy before reaching root. We can skip many levels between child and root and still retain same hierarchical name
- X.1, X.2, and X.3 are descendants of X and root and direct children of X
- Y.3.p is descendant of Y.3 and root and direct child of Y.3

96.2. Logger Inheritance

Each logger has a set of allowed properties. Each logger may define its own value for those properties, inherit the value of its parent, or be assigned a default (as in the case for root).

96.3. Logger Threshold Level Inheritance

The first inheritance property we will look at is a familiar one to you—severity threshold level. As the diagram shows

- root, loggerX, security, loggerY.3, loggerX.1 and loggerX.3 set an explicit value for their threshold
- loggerX.2 and loggerY.3.p inherit the threshold from their parent

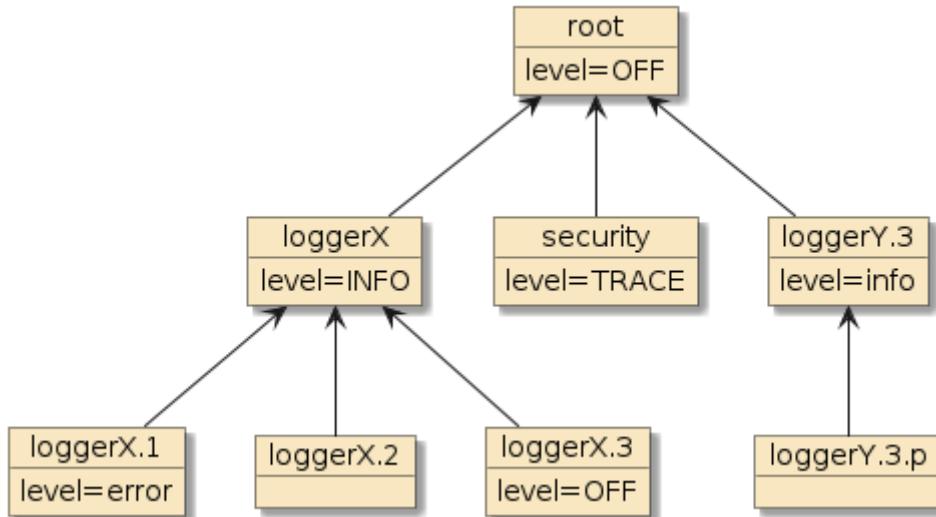


Figure 19. Logger Level Inheritance

96.4. Logger Effective Threshold Level Inheritance

The following table shows the specified and effective values applied to each logger for their threshold.

Table 6. Effective Logger Threshold Level

logger name	specified threshold	effective threshold
root	OFF	OFF
X	INFO	INFO
X.1	ERROR	ERROR
X.2		INFO
X.3	OFF	OFF
Y.3	WARN	WARN
Y.3.p		WARN
security	TRACE	TRACE

96.5. Example Logger Threshold Level Properties

These thresholds can be expressed in a property file.

application-tree.properties Snippet

```
logging.level.X=info
logging.level.X.1=error
logging.level.X.3=OFF
logging.level.security=trace
logging.level.Y.3=warn
logging.level.root=OFF
```

96.6. Example Logger Threshold Level Output

The output below demonstrates the impact of logging level inheritance from ancestors to descendants.

Effective Logger Threshold Level Output

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=tree

CONSOLE 05:58:14.956 INFO - X#run:25 X info
CONSOLE 05:58:14.959 WARN - X#run:26 X warn
CONSOLE 05:58:14.959 ERROR - X#run:27 X error
CONSOLE 05:58:14.960 ERROR - X.1#run:27 X.1 error ②
CONSOLE 05:58:14.960 INFO - X.2#run:25 X.2 info ①
CONSOLE 05:58:14.960 WARN - X.2#run:26 X.2 warn
CONSOLE 05:58:14.960 ERROR - X.2#run:27 X.2 error
CONSOLE 05:58:14.960 WARN - Y.3#run:26 Y.3 warn
CONSOLE 05:58:14.960 ERROR - Y.3#run:27 Y.3 error
CONSOLE 05:58:14.960 WARN - Y.3.p#run:26 Y.3.p warn ①
CONSOLE 05:58:14.961 ERROR - Y.3.p#run:27 Y.3.p error
CONSOLE 05:58:14.961 TRACE - security#run:23 security trace ③
CONSOLE 05:58:14.961 DEBUG - security#run:24 security debug
CONSOLE 05:58:14.962 INFO - security#run:25 security info
CONSOLE 05:58:14.962 WARN - security#run:26 security warn
CONSOLE 05:58:14.962 ERROR - security#run:27 security error
```

① X.2 and Y.3.p exhibit the same threshold level as their parents X (INFO) and Y.3 (WARN)

② X.1 (ERROR) and X.3 (OFF) override their parent threshold levels

③ security is writing all levels >= TRACE

Chapter 97. Appenders

Loggers generate [LoggerEvents](#) but do not directly log anything. Appenders are responsible for taking a [LoggerEvent](#) and producing a message to a log. There are many types of appenders. We have been working exclusively with a [ConsoleAppender](#) thus far but will work with some others before we are done. At this point — just know that a [ConsoleLogger](#) uses:

- an encoder to determine when to write messages to the log
- a layout to determine how to transform an individual [LoggerEvent](#) to a String
- a pattern when using a [PatternLayout](#) to define the transformation

97.1. Logger has N Appenders

Each of the loggers in our tree has the chance to have 0..N appenders.

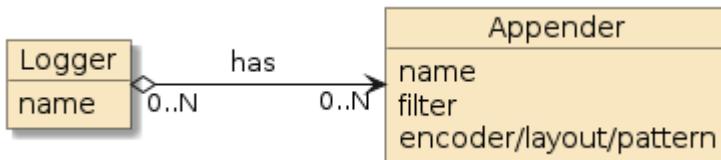


Figure 20. Logger / Appender Relationship

97.2. Logger Configuration Files

To date we have been able to work mostly with Spring Boot properties when using loggers. However, we will need to know a few things about the Logger Configuration File in order to define an appender and assign it to logger(s). We will start with how the logger configuration is found.

Logback and Log4J2 both use XML as their primary definition language. Spring Boot will automatically locate a well-known named configuration file in the root of the classpath:

- `logback.xml` or `logback-spring.xml` for Logback
- `log4j2.xml` or `log4j2-spring.xml` for Log4J2

[Spring Boot documentation](#) recommends using the `-spring.xml` suffixed files over the provider default named files in order for Spring Boot to assure that all documented features can be enabled. Alternately, we can explicitly specify the location using the `logging.config` property to reference anywhere in the classpath or file system.

application-tree.properties Reference

```
...  
logging.config=classpath:/logging-configs/tree/logback-spring.xml ①  
...
```

① an explicit property reference to the logging configuration file to use

97.3. Logback Root Configuration Element

The XML file has a root `configuration` element which contains details of the appender(s) and logger(s). See the [Spring Boot Configuration Documentation](#) and the [Logback Configuration Documentation](#) for details on how to configure.

logging-configs/tree/logback-spring.xml Configuration

```
<configuration debug="false"> ①  
  ...  
</configuration>
```

① `debug` attribute triggers [logback debug](#)

97.4. Retain Spring Boot Defaults

We will lose most/all of the default Spring Boot customizations for logging when we define our own custom logging configuration file. We can restore them by adding an `include`. This is that same file that we looked at earlier for the definition of `CONSOLE_LOG_PATTERN`.

logging-configs/tree/logback-spring.xml Retain Spring Boot Defaults

```
<configuration>  
  <!-- bring in Spring Boot defaults for Logback -->  
  <include resource="org/springframework/boot/logging/logback/defaults.xml"/>  
  ...  
</configuration>
```

97.5. Appender Configuration

Our example tree has three (3) appenders total. Each adds a literal string prefix so we know which appender is being called.

logging-configs/tree/logback-spring.xml Appenders

```
<!-- leverages what Spring Boot would have given us for console -->  
<appender name="console" class="ch.qos.logback.core.ConsoleAppender">  
  <encoder class="ch.qos.logback.classic.encoder.PatternLayoutEncoder"> ①  
    <pattern>CONSOLE  ${CONSOLE_LOG_PATTERN}</pattern> ②  
    <charset>utf8</charset>  
  </encoder>  
</appender>  
<appender name="X-appender" class="ch.qos.logback.core.ConsoleAppender">  
  <encoder>  
    <pattern>X      ${CONSOLE_LOG_PATTERN}</pattern>  
  </encoder>  
</appender>  
<appender name="security-appender" class="ch.qos.logback.core.ConsoleAppender">
```

```

<encoder>
    <pattern>SECURITY ${CONSOLE_LOG_PATTERN}</pattern>
</encoder>
</appender>

```

- ① PatternLayoutEncoder is the default encoder
 ② CONSOLE_PATTERN_LAYOUT is defined in included `defaults.xml`



This example forms the basis for demonstrating logger/appender assignment and appender additivity. `ConsoleAppender` is used in each case for ease of demonstration and not meant to depict a realistic configuration.

97.6. Appenders Attached to Loggers

The appenders are each attached to a single logger using the `appender-ref` element.

- console is attached to the root logger
- X-appender is attached to loggerX logger
- security-appender is attached to security logger

I am latching the two child appender assignments within an `appenders` profile to:

1. keep them separate from the earlier log level demo
2. demonstrate how to leverage Spring Boot extensions to build profile-based conditional logging configurations.

logging-configs/tree/logback-spring.xml Loggers

```

<springProfile name="appenders"> ①
    <logger name="X">
        <appender-ref ref="X-appender"/> ②
    </logger>

    <!-- this logger starts a new tree of appenders, nothing gets written to root
    logger -->
    <logger name="security" additivity="false">
        <appender-ref ref="security-appender"/>
    </logger>
</springProfile>

<root>
    <appender-ref ref="console"/>
</root>

```

- ① using Spring Boot Logback extension to only enable appenders when profile active
 ② appenders associated with logger using `appender-ref`

97.7. Appender Tree Inheritance

These appenders, in addition to level, are inherited from ancestor to descendant unless there is an override defined by the property `additivity=false`.

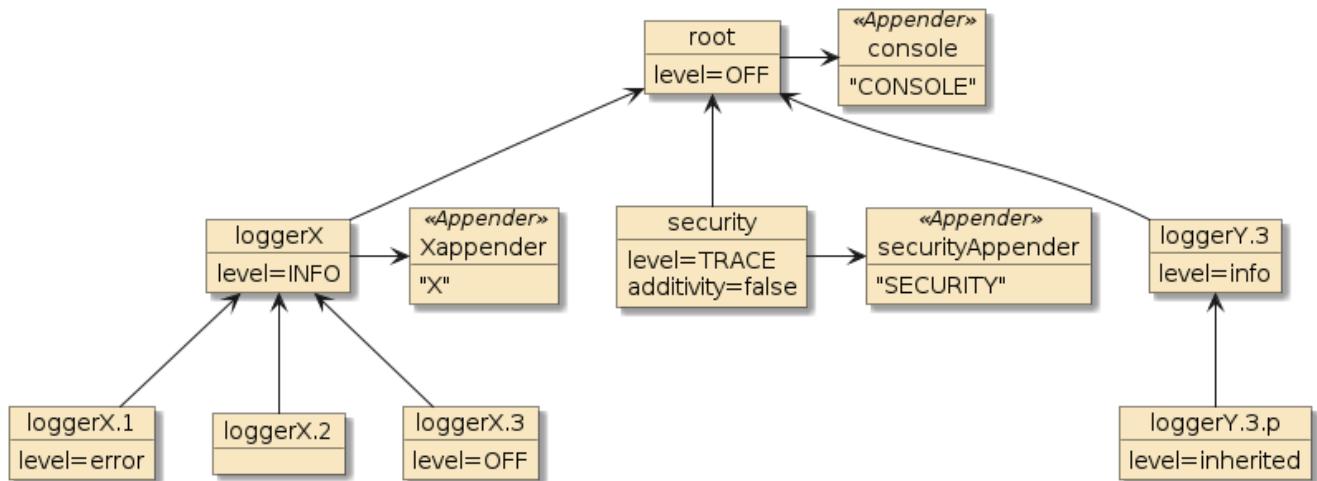


Figure 21. Example Logger Tree with Appenders

97.8. Appender Additivity Result

logger name	assigned threshold	assigned appender	effective threshold	effective appender
root	OFF	console	OFF	console
X	INFO	X-appender	INFO	console, X-appender
X.1	ERROR		ERROR	console, X-appender
X.2			INFO	console, X-appender
X.3	OFF		OFF	console, X-appender
Y.3	WARN		WARN	console
Y.3.p			WARN	console
security *additivity=false	TRACE	security-appender	TRACE	security-appender

97.9. Logger Inheritance Tree Output

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=tree,appenders
```

```
X      19:12:07.220  INFO - X#run:25 X info ①
CONSOLE 19:12:07.220  INFO - X#run:25 X info ①
X      19:12:07.224  WARN - X#run:26 X warn
```

```
CONSOLE 19:12:07.224  WARN - X#run:26 X warn
X       19:12:07.225 ERROR - X#run:27 X error
CONSOLE 19:12:07.225 ERROR - X#run:27 X error
X       19:12:07.225 ERROR - X.1#run:27 X.1 error
CONSOLE 19:12:07.225 ERROR - X.1#run:27 X.1 error
X       19:12:07.225 INFO  - X.2#run:25 X.2 info
CONSOLE 19:12:07.225 INFO  - X.2#run:25 X.2 info
X       19:12:07.225 WARN  - X.2#run:26 X.2 warn
CONSOLE 19:12:07.225 WARN  - X.2#run:26 X.2 warn
X       19:12:07.226 ERROR - X.2#run:27 X.2 error
CONSOLE 19:12:07.226 ERROR - X.2#run:27 X.2 error
CONSOLE 19:12:07.226  WARN - Y.3#run:26 Y.3 warn ②
CONSOLE 19:12:07.227 ERROR - Y.3#run:27 Y.3 error ②
CONSOLE 19:12:07.227  WARN - Y.3.p#run:26 Y.3.p warn
CONSOLE 19:12:07.227 ERROR - Y.3.p#run:27 Y.3.p error
SECURITY 19:12:07.227 TRACE - security#run:23 security trace ③
SECURITY 19:12:07.227 DEBUG - security#run:24 security debug ③
SECURITY 19:12:07.227 INFO  - security#run:25 security info ③
SECURITY 19:12:07.228  WARN - security#run:26 security warn ③
SECURITY 19:12:07.228 ERROR - security#run:27 security error ③
```

① log messages written to logger X and descendants are written to console and X-appender appenders

② log messages written to logger Y.3 and descendants are written only to console appender

③ log messages written to security logger are written only to security appender because of **additivity=false**

Chapter 98. Mapped Diagnostic Context

Thus far, we have been focusing on calls made within the code without much concern about the overall context in which they were made. In a multi-threaded, multi-user environment there is additional context information related to the code making the calls that we may want to keep track of—like userId and transactionId.

SLF4J and the logging implementations support the need for call context information through the use of [Mapped Diagnostic Context \(MDC\)](#). The `MDC` class is essentially a `ThreadLocal` map of strings that are assigned for the current thread. The values of the MDC are commonly set and cleared in container filters that fire before and after client calls are executed.

98.1. MDC Example

The following is an example where the `run()` method is playing the role of the container filter—setting and clearing the MDC. For this MDC map—I am setting a "user" and "requestId" key with the current user identity and a value that represents the request. The `doWork()` method is oblivious of the MDC and simply logs the start and end of work.

MDC Example

```
import org.slf4j.MDC;
...
public class MDCLogger implements CommandLineRunner {
    private static final String[] USERS = new String[]{"jim", "joe", "mary"};
    private static final SecureRandom r = new SecureRandom();

    @Override
    public void run(String... args) throws Exception {
        for (int i=0; i<5; i++) {
            String user = USERS[r.nextInt(USERS.length)];
            MDC.put("user", user); ①
            MDC.put("requestId", Integer.toString(r.nextInt(99999)));
            doWork();
            MDC.clear(); ②
            doWork();
        }
    }

    public void doWork() {
        log.info("starting work");
        log.info("finished work");
    }
}
```

① `run()` method simulates container filter setting context properties before call executed

② context properties removed after all calls for the context complete

98.2. MDC Example Pattern

To make use of the new "user" and "requestId" properties of the thread, we can add the `%mdc` (or `%X`) conversion word to the appender pattern as follows.

Adding MDC Properties to Pattern

```
#application-mdc.properties
logging.pattern.console=%date{HH:mm:ss.SSS} %-5level [%-9mdc{user:-anonymous}][%5mdc{requestId}] %logger{0} - %msg%n
```

- `%mdc{user:-anonymous}` - the identity of the user making the call or "anonymous" if not supplied
- `%mdc{requestId}` - the specific request made or blank if not supplied

98.3. MDC Example Output

The following is an example of running the MDC example. Users are randomly selected and work is performed for both identified and anonymous users. This allows us to track who made the work request and sort out the results of each work request.

MDC Example Output

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar
--spring.profiles.active=mdc

17:11:59.100 INFO [jim      ][61165] MDCLogger - starting work
17:11:59.101 INFO [jim      ][61165] MDCLogger - finished work
17:11:59.101 INFO [anonymous][      ] MDCLogger - starting work
17:11:59.101 INFO [anonymous][      ] MDCLogger - finished work
17:11:59.101 INFO [mary     ][ 8802] MDCLogger - starting work
17:11:59.101 INFO [mary     ][ 8802] MDCLogger - finished work
17:11:59.101 INFO [anonymous][      ] MDCLogger - starting work
17:11:59.101 INFO [anonymous][      ] MDCLogger - finished work
17:11:59.101 INFO [mary     ][86993] MDCLogger - starting work
17:11:59.101 INFO [mary     ][86993] MDCLogger - finished work
17:11:59.101 INFO [anonymous][      ] MDCLogger - starting work
17:11:59.101 INFO [anonymous][      ] MDCLogger - finished work
17:11:59.102 INFO [mary     ][67677] MDCLogger - starting work
17:11:59.102 INFO [mary     ][67677] MDCLogger - finished work
17:11:59.102 INFO [anonymous][      ] MDCLogger - starting work
17:11:59.102 INFO [anonymous][      ] MDCLogger - finished work
17:11:59.102 INFO [jim      ][25693] MDCLogger - starting work
17:11:59.102 INFO [jim      ][25693] MDCLogger - finished work
17:11:59.102 INFO [anonymous][      ] MDCLogger - starting work
17:11:59.102 INFO [anonymous][      ] MDCLogger - finished work
```



Like standard `ThreadLocal` variables, child threads do not inherit values of parent

thread. Each thread will maintain its own MDC properties.

98.4. Clearing MDC Context

We are responsible for setting the MDC context variables as well as clearing them when the work is complete.

- One way to do that is using a finally block and manually calling MDC.clear()

```
try {
    MDC.put("user", user); ①
    MDC.put("requestId", requestId);
    doWork();
} finally {
    MDC.clear();
}
```

- Another is by using a try-with-closable and have the properties automatically cleared when the try block finishes.

```
try (MDC.MDCCloseable mdc = MDC.putCloseable("user", user);
      MDC.MDCCloseable mdc = MDC.putCloseable("requestId", requestId) {
    doWork();
}
```

Chapter 99. Markers

SLF4J and the logging implementations support [markers](#). Unlike MDC data—which quietly sit in the background—markers are optionally supplied on a per-call basis. Markers have two primary uses

- trigger reporting events to appenders—e.g., flush log, send the e-mail
- implement additional severity levels—e.g., `log.warn(FLESH_WOUND, "come back here!")` versus `log.warn(FATAL, "ouch!!!")`^[1]



The additional functionality commonly is implemented through the use of filters assigned to appenders looking for these [Markers](#).



To me having triggers initiated by the logging statements does not sound appropriate (but still could be useful). However, when the thought of filtering comes up—I think of cases where we may want to better classify the subject(s) of the statement so that we have more to filter on when configuring appenders. More than once I have been in a situation where adjusting the verbosity of a single logger was not granular enough to provide an ideal result.

99.1. Marker Class

[Markers](#) have a single property called name and an optional collection of child [Markers](#). The name and collection properties allow the parent marker to represent one or more values. Appender filters test [Markers](#) using the `contains()` method to determine if the parent or any children are the targeted value.

[Markers](#) are obtained through the [MarkerFactory](#)—which caches the [Markers](#) it creates unless requested to make them detached so they can be uniquely added to separate parents.

99.2. Marker Example

The following simple example issues two log events. The first is without a [Marker](#) and the second with a [Marker](#) that represents the value [ALARM](#).

Marker Example

```
import org.slf4j.Marker;
import org.slf4j.MarkerFactory;
...
public class MarkerLogger implements CommandLineRunner {
    private static final Marker ALARM = MarkerFactory.getMarker("ALARM"); ①

    @Override
    public void run(String... args) throws Exception {
        log.warn("non-alarming warning statement"); ②
        log.warn(ALARM, "alarming statement"); ③
    }
}
```

```
    }  
}
```

- ① created single managed marker
- ② no marker added to logging call
- ③ marker added to logging call to trigger something special about this call

99.3. Marker Appender Filter Example

The Logback configuration has two appenders. The first appender—`alarms`—is meant to log only log events with an ALARM marker. I have applied the Logback-supplied `EvaluatorFilter` and `OnMarkerEvaluator` to eliminate any log events that do not meet that criteria.

Alarm Appender

```
<appender name="alarms" class="ch.qos.logback.core.ConsoleAppender">  
    <filter class="ch.qos.logback.core.filter.EvaluatorFilter">  
        <evaluator name="ALARM" class=  
            "ch.qos.logback.classic.boolex.OnMarkerEvaluator">  
            <marker>ALARM</marker>  
        </evaluator>  
        <onMatch>ACCEPT</onMatch>  
        <onMismatch>DENY</onMismatch>  
    </filter>  
    <encoder>  
        <pattern>%red(ALARM&gt;&gt;&gt; ${CONSOLE_LOG_PATTERN})</pattern>  
    </encoder>  
</appender>
```

The second appender—console—accepts all log events.

All Event Appender

```
<appender name="console" class="ch.qos.logback.core.ConsoleAppender">  
    <encoder>  
        <pattern>${CONSOLE_LOG_PATTERN}</pattern>  
    </encoder>  
</appender>
```

Both appenders are attached to the same root logger—which means that anything logged to the alarm appender will also be logged to the console appender.

Both Appenders added to root Logger

```
<configuration>  
    <include resource="org/springframework/boot/logging/logback/defaults.xml"/>  
    ...  
    <root>
```

```
<appender-ref ref="console"/>
<appender-ref ref="alarms"/>
</root>
</configuration>
```

99.4. Marker Example Result

The following shows the results of running the marker example — where both events are written to the console appender and only the log event with the **ALARM** Marker is written to the alarm appender.

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=markers

18:06:52.135 WARN  [] MarkerLogger - non-alarming warning statement ①
18:06:52.136 WARN  [ALARM] MarkerLogger - alarming statement ①
ALARM>>> 18:06:52.136 WARN  [ALARM] MarkerLogger - alarming statement ②
```

① non-ALARM and ALARM events are written to the console appender

② ALARM event is also written to alarm appender

[1] "what are markers in java logging frameworks and what is a reason to use them", Stack Overflow, 2019

Chapter 100. File Logging

Each topic and example so far has been demonstrated using the console because it is simple to demonstrate and to try out for yourself. However, once we get into more significant use of our application we are going to need to write this information somewhere to analyze later when necessary.

For that purpose, Spring Boot has a built-in appender ready to go for file logging. It is not active by default but all we have to do is specify the file name or path to trigger its activation.

Trigger FILE Appender

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=levels \
--logging.file.name="mylog.log" ①

$ ls ./mylog.log ②
./mylog.log
```

① adding this property adds file logging to default configuration

② this expressed logfile will be written to `mylog.log` in current directory

100.1. root Logger Appenders

As we saw earlier with appender additivity, multiple appenders can be associated with the same logger (root logger in this case). With the trigger property supplied, a file-based appender is added to the root logger to produce a log file in addition to our console output.

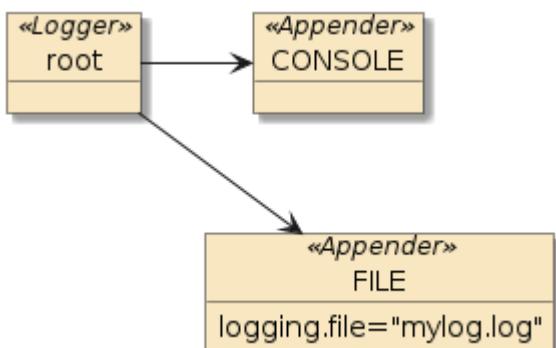


Figure 22. root Logger Appenders

100.2. FILE Appender Output

Under these simple conditions, a file is produced in the current directory with the specified `mylog.log` filename and the following contents.

FILE Appender File Output

```
$ cat mylog.log ①
2020-03-29 07:14:33.533 INFO 90958 --- [main] i.e.e.a.c.logging.levels.LoggerLevels
```

```

: info message
2020-03-29 07:14:33.542  WARN 90958 --- [main] i.e.e.a.c.logging.levels.LoggerLevels
: warn message
2020-03-29 07:14:33.542 ERROR 90958 --- [main] i.e.e.a.c.logging.levels.LoggerLevels
: error message

```

① written to file specified by `logging.file` property

The file and parent directories will be created if they do not exist. The default definition of the appender will append to an existing file if it already exists. Therefore—if we run the example a second time we get a second set of messages in the file.

FILE Appender Defaults to Append Mode

```

$ cat mylog.log
2020-03-29 07:14:33.533  INFO 90958 --- [main] i.e.e.a.c.logging.levels.LoggerLevels
: info message
2020-03-29 07:14:33.542  WARN 90958 --- [main] i.e.e.a.c.logging.levels.LoggerLevels
: warn message
2020-03-29 07:14:33.542 ERROR 90958 --- [main] i.e.e.a.c.logging.levels.LoggerLevels
: error message
2020-03-29 07:15:00.338  INFO 91090 --- [main] i.e.e.a.c.logging.levels.LoggerLevels
: info message ①
2020-03-29 07:15:00.342  WARN 91090 --- [main] i.e.e.a.c.logging.levels.LoggerLevels
: warn message
2020-03-29 07:15:00.342 ERROR 91090 --- [main] i.e.e.a.c.logging.levels.LoggerLevels
: error message

```

① messages from second execution appended to same log

100.3. Spring Boot FILE Appender Definition

If we take a look at the definition for [Spring Boot's Logback FILE Appender](#), we can see that it is a [Logback RollingFileAppender](#) with a [Logback SizeAndTimeBasedRollingPolicy](#).

Spring Boot Default FILE Appender Definition

```

<appender name="FILE"
    class="ch.qos.logback.core.rolling.RollingFileAppender"> ①
    <filter class="ch.qos.logback.classic.filter.ThresholdFilter">
        <level>${FILE_LOG_THRESHOLD}</level>
    </filter>
    <encoder>
        <pattern>${FILE_LOG_PATTERN}</pattern>
        <charset>${FILE_LOG_CHARSET}</charset>
    </encoder>
    <file>${LOG_FILE}</file>
    <rollingPolicy class="ch.qos.logback.core.rolling.SizeAndTimeBasedRollingPolicy">
        ②
        <fileNamePattern>${LOGBACK_ROLLINGPOLICY_FILE_NAME_PATTERN}-

```

```

${LOG_FILE}.%d{yyyy-MM-dd}.%i.gz}</fileNamePattern>
    <cleanHistoryOnStart>${LOGBACK_ROLLINGPOLICY_CLEAN_HISTORY_ON_START:-false}</cleanHistoryOnStart>
        <maxFileSize>${LOGBACK_ROLLINGPOLICY_MAX_FILE_SIZE:-10MB}</maxFileSize>
        <totalSizeCap>${LOGBACK_ROLLINGPOLICY_TOTAL_SIZE_CAP:-0}</totalSizeCap>
        <maxHistory>${LOGBACK_ROLLINGPOLICY_MAX_HISTORY:-7}</maxHistory>
    </rollingPolicy>
</appender>

```

- ① performs file rollover functionality based on configured policy
- ② specifies policy and policy configuration to use

100.4. RollingFileAppender

The [Logback RollingFileAppender](#) will:

- write log messages to a specified file — and at some point, switch to writing to a different file
- use a triggering policy to determine the point in which to switch files (i.e., "when it will occur")
- use a rolling policy to determine how the file switchover will occur (i.e., "what will occur")
- use a single policy for both if the rolling policy implements both policy interfaces
- use file append mode by default



The rollover settings/state is evaluated no sooner than once a minute. If you set the maximum sizes to small amounts and log quickly for test/demonstration purposes, you will exceed your defined size limits until the recheck timeout has expired.

100.5. SizeAndTimeBasedRollingPolicy

The [Logback SizeAndTimeBasedRollingPolicy](#) will:

- trigger a file switch when the current file reaches a maximum size
- trigger a file switch when the granularity of the primary date (%d) pattern in the file path/name would rollover to a new value
- supply a name for the old/historical file using a mandatory date (%d) pattern and index (%i)
- define a maximum number of historical files to retain
- define a total size to allocate to current and historical files

100.6. FILE Appender Properties

name	description	default
logging.file.path	full or relative path of directory written to — ignored when logging.file.name provided	.

name	description	default
logging.file.name	full or relative path of filename written to — may be manually built using <code>/logging.file.path</code>	<code> \${logging.file.path}/spring.log</code>
logging.pattern.rolling-file-name	pattern expression for historical file — must include a date and index — may express compression	<code> \${logging.file.name}.%d{yyyy-MM-dd}.%i.gz</code>
logging.logback.rollingpolicy.max-file-size	maximum size of log before changeover — must be less than <code>total-size-cap</code>	10MB
logging.logback.rollingpolicy.max-history	maximum number of historical files to retain when changing over because of date criteria	7
logging.logback.rollingpolicy.total-size-cap	maximum amount of total space to consume — must be greater than <code>max-size</code>	(no limit)



If file logger property value is invalid, the application will run without the FILE appender activated.

100.7. logging.file.path

If we specify only the `logging.file.path`, the filename will default to `spring.log` and will be written to the directory path we supply.

logging.file.path Example

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--logging.file.path=target/logs ①
...
$ ls target/logs ②
spring.log
```

① specifying `logging.file.path` as `target/logs`

② produces a `spring.log` in that directory

100.8. logging.file.name

If we specify only the `logging.file.name`, the file will be written to the filename and directory we explicitly supply.

logging.file.name Example

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--logging.file.name=target/logs/mylog.log ①
...
$ ls target/logs ②
mylog.log
```

① specifying a `logging.file.name`

② produces a logfile with that path and name

100.9. `logging.logback.rollingpolicy.max-file-size` Trigger

One trigger for changing over to the next file is `logging.logback.rollingpolicy.max-file-size`. Note this property is Logback-specific. The condition is satisfied when the current logfile reaches this value. The default is 10MB.

The following example changes that to 9400 Bytes. Once each instance of `logging.file.name` reached the `logging.logback.rollingpolicy.max-file-size`, it is compressed and moved to a filename with the pattern from `logging.pattern.rolling-file-name`. I picked 9400 Bytes based on the fact the application wrote 4800 Bytes each minute. The file size would be evaluated each minute and exceed the limit every 2 minutes. Notice the uncompressed, archived files are at least 9400 Bytes and 2 minutes apart.



Evaluation is no more often than 1 minute.

logging.logback.rollingpolicy.max-file-size Example

```
java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=rollover \
--logging.file.name=target/logs/mylog.log \
--logging.pattern.rolling-file-name='${logging.file.name}.%d{yyyy-MM-dd}.%i.log' \
--logging.logback.rollingpolicy.max-file-size=9400B ①
...
ls -ltrh target/logs/
-rw-r--r-- 1 jim staff 9.4K Aug 9 11:29 mylog.log.2024-08-09.0.log ③
-rw-r--r-- 1 jim staff 9.5K Aug 9 11:31 mylog.log.2024-08-09.1.log ②
-rw-r--r-- 1 jim staff 1.6K Aug 9 11:31 mylog.log ①
```

① `logging.logback.rollingpolicy.max-file-size` limits the size of the current logfile

② historical logfiles renamed according to `logging.pattern.rolling-file-name` pattern

③ file size is evaluated each minute and archived when satisfied

100.10. logging.pattern.rolling-file-name

There are several aspects of `logging.pattern.rolling-file-name` to be aware of

- `%d` timestamp pattern and `%i` index are required. The FILE appender will either be disabled (for `%i`) or the application startup will fail (for `%d`) if not specified
- the timestamp pattern directly impacts changeover when there is a value change in the result of applying the timestamp pattern. Many of my examples here use a pattern that includes `HH:mm:ss` just for demonstration purposes. A more common pattern would be by date only.
- the index is used when the `logging.logback.rollingpolicy.max-file-size` triggers the changeover and we already have a historical name with the same timestamp.
- the number of historical files is throttled using `logging.logback.rollingpolicy.max-history` only when index is used and not when file changeover is due to `logging.logback.rollingpolicy.max-file-size`
- the historical file will be compressed if `gz` is specified as the suffix

100.11. Timestamp Rollover Example

The following example shows the file changeover occurring because the evaluation of the `%d` template expression within `logging.pattern.rolling-file-name` changing. The historical file is left uncompressed because the `logging.pattern.rolling-file-name` does not end in `gz`.

Timestamp Rollover Example

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=rollover \
--logging.file.name=target/logs/mylog.log \
--logging.pattern.rolling-file-name='${logging.file.name}.\%d{yyyy-MM-dd
-HH:mm:ss}.\%i'.log ①
...
$ ls -ltrh target/logs

total 64
-rw-r--r-- 1 jim staff 79B Aug 9 12:04 mylog.log.2024-08-09-12:04:54.0.log
-rw-r--r-- 1 jim staff 79B Aug 9 12:04 mylog.log.2024-08-09-12:04:55.0.log
-rw-r--r-- 1 jim staff 79B Aug 9 12:04 mylog.log.2024-08-09-12:04:56.0.log
-rw-r--r-- 1 jim staff 79B Aug 9 12:04 mylog.log.2024-08-09-12:04:57.0.log
-rw-r--r-- 1 jim staff 79B Aug 9 12:04 mylog.log.2024-08-09-12:04:58.0.log
-rw-r--r-- 1 jim staff 79B Aug 9 12:04 mylog.log.2024-08-09-12:04:59.0.log
-rw-r--r-- 1 jim staff 79B Aug 9 12:05 mylog.log.2024-08-09-12:05:00.0.log
-rw-r--r-- 1 jim staff 79B Aug 9 12:05 mylog.log

$ file target/logs/mylog.log.2024-08-09-12\:04\:54.0.log ②
target/logs/mylog.log.2024-08-09-12:04:54.0.log: ASCII text
```

① `logging.pattern.rolling-file-name` pattern triggers changeover at the seconds boundary

- ② historical logfiles are left uncompressed because of `.log` name suffix specified



Using a date pattern to include minutes and seconds is just for demonstration and learning purposes. Most patterns would be daily.

100.12. History Compression Example

The following example is similar to the previous one with the exception that the `logging.pattern.rolling-file-name` ends in `gz` — triggering the historical file to be compressed.

History Compression Example

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=rollover \
--logging.file.name=target/logs/mylog.log \
--logging.pattern.rolling-file-name='${logging.file.name}.%d{yyyy-MM-dd-HH:mm}.%i'.gz
①
...
$ ls -ltrh target/logs

total 48
-rw-r--r-- 1 jim staff 193B Aug 9 13:39 mylog.log.2024-08-09-13:38.0.gz ①
-rw-r--r-- 1 jim staff 534B Aug 9 13:40 mylog.log.2024-08-09-13:39.0.gz
-rw-r--r-- 1 jim staff 540B Aug 9 13:41 mylog.log.2024-08-09-13:40.0.gz
-rw-r--r-- 1 jim staff 528B Aug 9 13:42 mylog.log.2024-08-09-13:41.0.gz
-rw-r--r-- 1 jim staff 539B Aug 9 13:43 mylog.log.2024-08-09-13:42.0.gz
-rw-r--r-- 1 jim staff 1.7K Aug 9 13:43 mylog.log

$ file target/logs/mylog.log.2024-08-09-13:38.0.gz
target/logs/mylog.log.2024-08-09-13:38.0.gz: gzip compressed data, original size
modulo 2^32 1030
```

- ① historical logfiles are compressed when pattern uses a `.gz` suffix

100.13. logging.logback.rollingpolicy.max-history Example

`logging.logback.rollingpolicy.max-history` will constrain the number of files created for independent timestamps. In the example below, I constrained the limit to 2. Note that the `logging.logback.rollingpolicy.max-history` property does not seem to apply to files terminated because of size. For that, we can use `logging.file.total-size-cap`.

Max History Example

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=rollover \
--logging.file.name=target/logs/mylog.log \
--logging.pattern.rolling-file-name='${logging.file.name}.%d{yyyy-MM-dd}'
```

```

-HH:mm:ss}.\%i'.log \
--logging.logback.rollingpolicy.max-history=2
...
$ ls -ltrh target/logs

total 24
-rw-r--r-- 1 jim staff 80B Aug 9 12:15 mylog.log.2024-08-09-12:15:31.0.log ①
-rw-r--r-- 1 jim staff 80B Aug 9 12:15 mylog.log.2024-08-09-12:15:32.0.log ①
-rw-r--r-- 1 jim staff 80B Aug 9 12:15 mylog.log

```

① specifying `logging.logback.rollingpolicy.max-history` limited number of historical logfiles. Oldest files exceeding the criteria are deleted.

100.14. logging.logback.rollingpolicy.total-size-cap Index Example

The following example triggers file changeover every 1000 Bytes and makes use of the index because we encounter multiple changes per timestamp pattern. The files are aged-off at the point where total size for all logs reaches `logging.logback.rollingpolicy.total-size-cap`. Thus historical files with indexes 1 thru 9 have been deleted at this point in time in order to stay below the file size limit.

Total Size Limit (with Index) Example

```

$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=rollover \
--logging.file.name=target/logs/mylog.log \
--logging.logback.rollingpolicy.max-file-size=1000 \
--logging.pattern.rolling-file-name='${logging.file.name}.\%d{yyyy-MM-dd}.\%i'.log \
--logging.logback.rollingpolicy.total-size-cap=10000 ①
...
$ ls -ltr target/logs

total 40 ②
-rw-r--r-- 1 jim staff 4.7K Aug 9 12:37 mylog.log.2024-08-09.10.log ①
-rw-r--r-- 1 jim staff 4.7K Aug 9 12:38 mylog.log.2024-08-09.11.log ①
-rw-r--r-- 1 jim staff 2.7K Aug 9 12:39 mylog.log ①

```

① `logging.logback.rollingpolicy.total-size-cap` constrains current plus historical files retained

② historical files with indexes 1 thru 9 were deleted to stay below file size limit

100.15. logging.logback.rollingpolicy.total-size-cap no Index Example

The following example triggers file changeover every second and makes no use of the index because the timestamp pattern is so granular that `max-size` is not reached before the timestamp changes the base. As with the previous example, the files are also aged-off when the total byte count reaches `logging.logback.rollingpolicy.total-size-cap`.

Total Size Limit (without Index) Example

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=rollover \
--logging.file.name=target/logs/mylog.log \
--logging.logback.rollingpolicy.max-file-size=100 \
--logging.pattern.rolling-file-name='${logging.file.name}.\%d{yyyy-MM-dd
-HH:mm:ss}.\%i'.log \
--logging.logback.rollingpolicy.max-history=200 \
--logging.logback.rollingpolicy.total-size-cap=500 ①
...
$ ls -ltrh target/logs

total 56
-rw-r--r-- 1 jim staff 79B Aug 9 12:44 mylog.log.2024-08-09-12:44:33.0.log ①
-rw-r--r-- 1 jim staff 79B Aug 9 12:44 mylog.log.2024-08-09-12:44:34.0.log ①
-rw-r--r-- 1 jim staff 79B Aug 9 12:44 mylog.log.2024-08-09-12:44:35.0.log ①
-rw-r--r-- 1 jim staff 79B Aug 9 12:44 mylog.log.2024-08-09-12:44:36.0.log ①
-rw-r--r-- 1 jim staff 79B Aug 9 12:44 mylog.log.2024-08-09-12:44:37.0.log ①
-rw-r--r-- 1 jim staff 79B Aug 9 12:44 mylog.log.2024-08-09-12:44:38.0.log ①
-rw-r--r-- 1 jim staff 80B Aug 9 12:44 mylog.log ①
```

① `logging.logback.rollingpolicy.total-size-cap` constrains current plus historical files retained



The `logging.logback.rollingpolicy.total-size-cap` value—if specified—must be larger than the `logging.logback.rollingpolicy.max-file-size` constraint. Otherwise the file appender will not be activated.

Chapter 101. Custom Configurations

At this point, you should have a good foundation in logging and how to get started with a decent logging capability and understand how the default configuration can be modified for your immediate and profile-based circumstances. For cases when this is not enough, know that:

- detailed XML Logback and Log4J2 configurations can be specified—which allows the definition of loggers, appenders, filters, etc. of nearly unlimited power
- Spring Boot provides include files that can be used as a starting point for defining the custom configurations without giving up most of what Spring Boot defines for the default configuration

101.1. Logback Configuration Customization

Although Spring Boot actually performs much of the configuration manually through [code](#), a set of XML includes are supplied that simulate most of what that setup code performs. We can perform the following steps to create a custom Logback configuration.

- create a `logback-spring.xml` file with a parent `configuration` element
 - place in root of application archive (i.e., `src/main/resources` of source tree)
- include one or more of the provided XML includes

101.2. Provided Logback Includes

- `defaults.xml` - defines the logging configuration defaults we have been working with
- `base.xml` - defines root logger with CONSOLE and FILE appenders we have discussed
 - puts you at the point of the out-of-the-box configuration
- `console-appender.xml` - defines the `CONSOLE` appender we have been working with
 - uses the `CONSOLE_LOG_PATTERN`
- `file-appender.xml` - defines the `FILE` appender we have been working with
 - uses the `RollingFileAppender` with `FILE_LOG_PATTERN` and `SizeAndTimeBasedRollingPolicy`

These files provide an XML representation of what Spring Boot configures with straight Java code. There are minor differences (e.g., enable/disable FILE Appender) between using the supplied XML files and using the out-of-the-box defaults.



101.3. Customization Example: Turn off Console Logging

The following is an example custom configuration where we wish to turn off console logging and only rely on the logfiles. This result is essentially a copy/edit of the supplied `base.xml`.

```
<!-- logging-configs/no-console/logback-spring.xml ①
    Example Logback configuration file to turn off CONSOLE Appender and retain all
other
    FILE Appender default behavior.

-->
<configuration>
    <include resource="org/springframework/boot/logging/logback/defaults.xml"/> ②
    <property name="LOG_FILE" value="${LOG_FILE:-${LOG_PATH:-${LOG_TEMP:-${java.io.tmpdir:-/tmp}}}}/spring.log"/> ③
    <include resource="org/springframework/boot/logging/logback/file-appender.xml"/> ④

    <root>
        <appender-ref ref="FILE"/> ⑤
    </root>
</configuration>
```

① a logback-spring.xml file has been created to host the custom configuration

② the standard Spring Boot defaults are included

③ LOG_FILE defined using the original expression from Spring Boot `base.xml`

④ the standard Spring Boot FILE appender is included

⑤ only the FILE appender is assigned to our logger(s)

101.4. LOG_FILE Property Definition

The only complicated part is what I copy/pasted from `base.xml` to express the `LOG_FILE` property used by the included FILE appender:

LOG_FILE Property Definition

```
<property name="LOG_FILE"
value="${LOG_FILE:-${LOG_PATH:-${LOG_TEMP:-${java.io.tmpdir:-/tmp}}}}/spring.log"/>
```

- use the value of `LOG_FILE` if that is defined
- otherwise use the filename `spring.log` and for the path
 - use `LOG_PATH` if that is defined
 - otherwise use `LOG_TEMP` if that is defined
 - otherwise use `java.io.tmpdir` if that is defined
 - otherwise use `/tmp`

101.5. Customization Example: Leverage Restored Defaults

Our first execution uses all defaults and is written to `${java.io.tmpdir}/spring.log`

Example with Default Logfile

```
java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--logging.config=src/main/resources/logging-configs/no-console/logback-spring.xml
(no console output)

$ ls -ltr $TMPDIR/spring.log ①
-rw-r--r-- 1 jim staff 67238 Apr 2 06:42
/var/folders/zm/cskr47zn0yjd0zwkn870y5sc0000gn/T//spring.log
```

① logfile written to restored default `${java.io.tmpdir}/spring.log`

101.6. Customization Example: Provide Override

Our second execution specified an override for the logfile to use. This is expressed exactly as we did earlier with the default configuration.

Example with Specified Logfile

```
java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--logging.config=src/main/resources/logging-configs/no-console/logback-spring.xml \
--logging.file.name="target/logs/mylog.log" ②
(no console output)

$ ls -ltr target/logs ①

total 136
-rw-r--r-- 1 jim staff 67236 Apr 2 06:46 mylog.log ①
```

① logfile written to `target/logs/mylog.log`

② defined using `logging.file.name`

Chapter 102. Spring Profiles

Spring Boot extends the logback.xml capabilities to allow us to easily take advantage of profiles. Any of the elements within the configuration file can be wrapped in a `springProfile` element to make their activation depend on the profile value.

Example Profile Use

```
<springProfile name="appenders"> ①
  <logger name="X">
    <appender-ref ref="X-appender"/>
  </logger>

  <!-- this logger starts a new tree of appenders, nothing gets written to root
logger -->
  <logger name="security" additivity="false">
    <appender-ref ref="security-appender"/>
  </logger>
</springProfile>
```

① elements are activated when `appenders` profile is activated

See [Profile-Specific Configuration](#) for more examples involving multiple profile names and boolean operations.

Chapter 103. Summary

In this module we:

- made a case for the value of logging
- demonstrated how logging frameworks are much better than `System.out` logging techniques
- discussed the different interface, adapter, and implementation libraries involved with Spring Boot logging
- learned how the interface of the logging framework is separate from the implementation
- learned to log information at different severity levels using loggers
- learned how to write logging statements that can be efficiently executed when disabled
- learned how to establish a hierarchy of loggers
- learned how to configure appenders and associate with loggers
- learned how to configure pattern layouts
- learned how to configure the FILE Appender
- looked at additional topics like Mapped Data Context (MDC) and Markers that can augment standard logging events

We covered the basics in great detail so that you understood the logging framework, what kinds of things are available to you, how it was doing its job, and how it could be configured. However, we still did not cover everything. For example, we left topics like accessing and viewing logs within a distributed environment, structured appender formatters (e.g., JSON), etc.. It is important for you to know that this lesson placed you at a point where those logging extensions can be implemented by you in a straight forward manner.

Testing

copyright © 2024 jim stafford (jim.stafford@jhu.edu)

Chapter 104. Introduction

104.1. Why Do We Test?

- demonstrate capability?
- verify/validate correctness?
- find bugs?
- aid design?
- more ...?

There are many great reasons to incorporate software testing into the application lifecycle. There is no time too early to start.

104.2. What are Test Levels?

- [Unit Testing](#) - verifies a specific area of code
- [Integration Testing](#) - any type of testing focusing on interface between components
- [System Testing](#) — tests involving the complete system
- [Acceptance Testing](#) — normally conducted as part of a contract sign-off

It would be easy to say that our focus in this lesson will be on unit and integration testing. However, there are some aspects of system and acceptance testing that are applicable as well.

104.3. What are some Approaches to Testing?

- [Static Analysis](#) — code reviews, syntax checkers
- [Dynamic Analysis](#) — takes place while code is running
- [White-box Testing](#) — makes use of an internal perspective
- [Black-box Testing](#) — makes use of only what the item is required to do
- [Many more ...](#)

In this lesson we will focus on dynamic analysis testing using both black-box interface contract testing and white-box implementation and collaboration testing.

104.4. Goals

The student will learn:

- to understand the testing frameworks bundled within Spring Boot Test Starter
- to leverage test cases and test methods to automate tests performed
- to leverage assertions to verify correctness

- to integrate mocks into test cases
- to implement unit integration tests within Spring Boot
- to express tests using Behavior-Driven Development (BDD) acceptance test keywords
- to automate the execution of tests using Maven
- to augment and/or replace components used in a unit integration test

104.5. Objectives

At the conclusion of this lecture and related exercises, the student will be able to:

1. write a test case and assertions using "Vintage" JUnit 4 constructs
2. write a test case and assertions using JUnit 5 "Jupiter" constructs
3. leverage alternate (JUnit, Hamcrest, AssertJ, etc.) assertion libraries
4. implement a mock (using Mockito) into a JUnit unit test
 - a. define custom behavior for a mock
 - b. capture and inspect calls made to mocks by subjects under test
5. implement BDD acceptance test keywords into Mockito & AssertJ-based tests
6. implement unit integration tests using a Spring context
7. implement (Mockito) mocks in Spring context for unit integration tests
8. augment and/or override Spring context components using `@TestConfiguration`
9. execute tests using Maven Surefire plugin

Chapter 105. Test Constructs

At the heart of testing, we want to

- establish a subject under test
- establish a context in which to test that subject
- perform actions on the subject
- evaluate the results

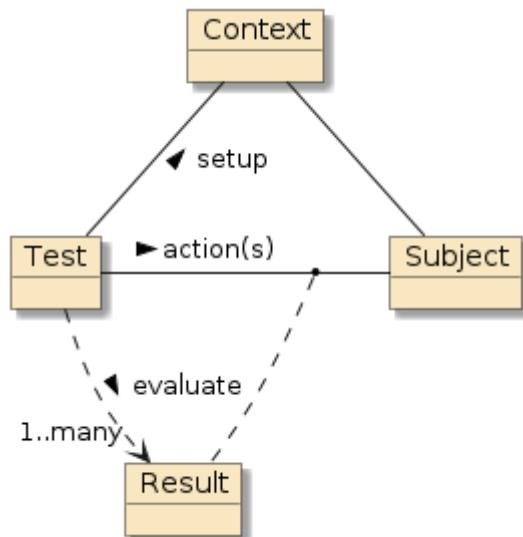


Figure 23. Basic Test Concepts

Subjects can vary in scope depending on the type of our test. Unit testing will have class and method-level subjects. Integration tests can span multiple classes/components—whether vertically (e.g., front-end request to database) or horizontally (e.g., peers).

105.1. Automated Test Terminology

Unfortunately, you will see the terms "unit" and "integration" used differently as we go through the testing topics and span tooling. There is a conceptual way of thinking of testing and a technical way of how to manage testing to be concerned with when seeing these terms used:

Conceptual - At a conceptual level, we simply think of unit tests dealing with one subject at a time and involve varying levels of simulation around them in order to test that subject. We conceptually think of integration tests at the point where multiple real components are brought together to form the overall set of subjects—whether that be vertical (e.g., to the database and back) or horizontal (e.g., peer interactions) in nature.

Test Management - At a test management level, we have to worry about what it takes to spin up and shutdown resources to conduct our testing. Build systems like Maven refer to unit tests as anything that can be performed within a single JVM and integration tests as tests that require managing external resources (e.g., start/stop web server). Maven runs these tests in different phases—executing unit tests first with the [Surefire plugin](#) and integration tests last with the [Failsafe plugin](#).

105.2. Maven Test Types

Maven runs these tests in different phases—executing unit tests first with the [Surefire plugin](#) and integration tests last with the [Failsafe plugin](#). By default, Surefire will [locate unit tests](#) starting with

"Test" or ending with "Test", "Tests", or "TestCase". Failsafe will [locate integration tests](#) starting with "IT" or ending with "IT" or "ITCase".

105.3. Test Naming Conventions

Neither tools like JUnit or the IDEs care how classes are named. However, since our goal is to eventually check these tests in with our source code and run them in an automated manner—we will have to pay early attention to Maven Surefire and Failsafe naming rules while we also address the conceptual aspects of testing.

105.4. Lecture Test Naming Conventions

I will try to use the following terms to mean the following:

- Unit Test - conceptual unit test focused on a limited subject and will use the suffix "Test". These will generally be run without a Spring context and will be picked up by Maven Surefire.
- Unit Integration Test - conceptual integration test (vertical or horizontal) runnable within a single JVM and will use the suffix "NTest". This will be picked up by Maven Surefire and will likely involve a Spring context.
- External Integration Test - conceptual integration test (vertical or horizontal) requiring external resource management and will use the suffix "IT". This will be picked up by Maven Failsafe. These will always have Spring context(s) running in one or more JVMs. These will sometimes be termed as "Maven Integration Tests" or "Failsafe Integration Tests".

That means to not be surprised to see a conceptual integration test bringing multiple real components together to be executed during the Maven Surefire test phase if we can perform this testing without the resource management of external processes.

Chapter 106. Spring Boot Starter Test Frameworks

We want to automate tests as much as possible and can do that with many of the Spring Boot testing options made available using the `spring-boot-starter-test` dependency. This single dependency defines transitive dependencies on several powerful, state of the art as well as legacy, testing frameworks. These dependencies are only used during builds and not in production—so we assign a scope of `test` to this dependency.

pom.xml spring-boot-test-starter Dependency

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope> ①
</dependency>
```

① dependency scope is `test` since these dependencies are not required to run outside of build environment

106.1. Spring Boot Starter Transitive Dependencies

If we take a look at the transitive dependencies brought in by `spring-boot-test-starter`, we see a wide array of choices pre-integrated.

spring-boot-starter-test Transitive Dependencies (reorganized)

```
[INFO] +- org.springframework.boot:spring-boot-starter-test:jar:3.3.2:test
[INFO] |   +- org.springframework.boot:spring-boot-test:jar:3.3.2:test
[INFO] |   +- org.springframework.boot:spring-boot-test-autoconfigure:jar:3.3.2:test
[INFO] |   +- org.springframework:spring-test:jar:6.1.11:test

[INFO] |   +- org.junit.jupiter:junit-jupiter:jar:5.10.3:test
[INFO] |   |   +- org.junit.jupiter:junit-jupiter-api:jar:5.10.3:test
[INFO] |   |   +- org.junit.jupiter:junit-jupiter-params:jar:5.10.3:test
[INFO] |   |   \- org.junit.jupiter:junit-jupiter-engine:jar:5.10.3:test

[INFO] |   +- org.assertj:assertj-core:jar:3.25.3:test
[INFO] |   |   \- net.bytebuddy:byte-buddy:jar:1.14.18:test
[INFO] |   +- org.hamcrest:hamcrest:jar:2.2:test
[INFO] +- org.exparity:hamcrest-date:jar:2.0.8:test
[INFO] |   \- org.hamcrest:hamcrest-core:jar:2.2:test

[INFO] |   +- org.mockito:mockito-core:jar:5.11.0:test
[INFO] |   |   +- net.bytebuddy:byte-buddy-agent:jar:1.14.18:test
[INFO] |   |   \- org.objenesis:objenesis:jar:3.3:test
[INFO] |   +- org.mockito:mockito-junit-jupiter:jar:5.11.0:test
```

```
[INFO] | +- com.jayway.jsonpath:json-path:jar:2.9.0:test
[INFO] | | \- org.slf4j:slf4j-api:jar:2.0.13:compile

[INFO] | +- org.skyscreamer:jsonassert:jar:1.5.3:test
[INFO] | \- org.xmlunit:xmlunit-core:jar:2.9.1:test

[INFO] \- org.junit.vintage:junit-vintage-engine:jar:5.10.3:test
[INFO]     +- org.junit.platform:junit-platform-engine:jar:1.10.3:test
[INFO]     +- junit:junit:jar:4.13.2:test
...

```

106.2. Transitive Dependency Test Tools

At a high level:

- **spring-boot-test-autoconfigure** - contains many auto-configuration classes that detect test conditions and configure common resources for use in a test mode
- **junit** - required to run the JUnit tests
- **hamcrest** - required to implement Hamcrest test assertions
- **assertj** - required to implement AssertJ test assertions
- **mockito** - required to implement Mockito mocks
- **jsonassert** - required to write flexible assertions for JSON data
- **jsonpath** - used to express paths within JSON structures
- **xmlunit** - required to write flexible assertions for XML data

In the rest of this lesson, I will be describing how JUnit, the assertion libraries, Mockito and Spring Boot play a significant role in unit and integration testing.

Chapter 107. JUnit Background

JUnit is a test framework that has been around for many years (I found [first commit in git](#) from Dec 3, 2000). The test framework was [originated by Kent Beck and Erich Gamma](#) during a plane ride they shared in 1997. Its basic structure is centered around:

- **tests** that perform actions on the subjects within a given context and assert proper results
- **test cases** that group tests and wrap in a set of common setup and teardown steps
- **test suites** that provide a way of grouping certain tests



Test Suites are not as pervasive as test cases and tests

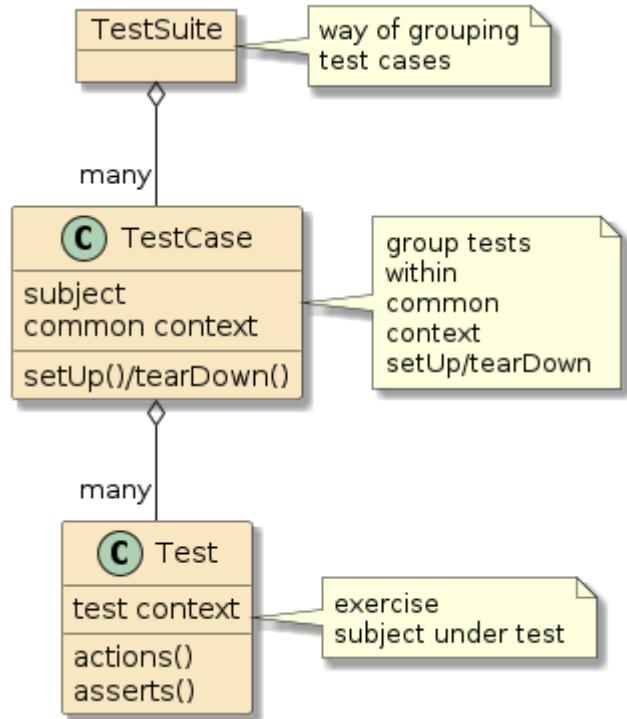


Figure 24. Basic JUnit Test Framework Constructs

These constructs have gone through evolutionary changes in Java—to include annotations in Java 5 and lambda functions in Java 8—which have provided substantial API changes in Java frameworks.

- annotations added in Java 5 permitted frameworks to move away from inheritance-based approaches—with specifically named methods (JUnit 3.8) and to leverage annotations added to classes and methods (JUnit 4)

JUnit 3.8 Test Case—Inheritance-based

```
public class MathTest extends TestCase {
    protected void setUp() { } ①
    protected void tearDown() { } ①
    public void testAdd() { ②
        assertEquals(4, 2+2);
    }
}
```

① `setUp()` and `tearDown()` are method overrides of base class `TestCase`

② all test methods were required to start with word `test`

- lambda functions (JUnit 5/Jupiter) added in Java 8 permit the flexible expression of code blocks that can extend the behavior of provided functionality without requiring verbose subclassing

JUnit 4/Vintage Assertions

```
assertEquals(5, 2+2); //fails here
assertEquals(3, 2+2); //not eval ①
assertEquals(String.format("try%d", 2),
    4, 2+2); ②
```

① evaluation will stop at first failure

② descriptions were first parameter and always evaluated

JUnit 4 Test Case—Annotation-based

```
public class MathTest {
    @Before
    public void setup() { } ①
    @After
    public void teardown() { } ①
    @Test
    public void add() { ①
        assertEquals(4, 2+2);
    }
}
```

① public methods found by annotation—no naming requirement

JUnit 5/Jupiter Lambda Assertions

```
assertAll//all get eval and reported ①
    () -> assertEquals(5, 2+2),
    () -> assertEquals(3, 2+2),
    () -> assertEquals(4, 2+2, ② ③
        ()->String.format("try%d", 2))
);
```

① all assertions can be evaluated

② descriptions are now last argument

③ only evaluated if fail with lambdas

107.1. JUnit 5 Evolution

The success and simplicity of JUnit 4 made it hard to incorporate new features. JUnit 4 was a single module/JAR and everything that used JUnit leveraged that single jar.

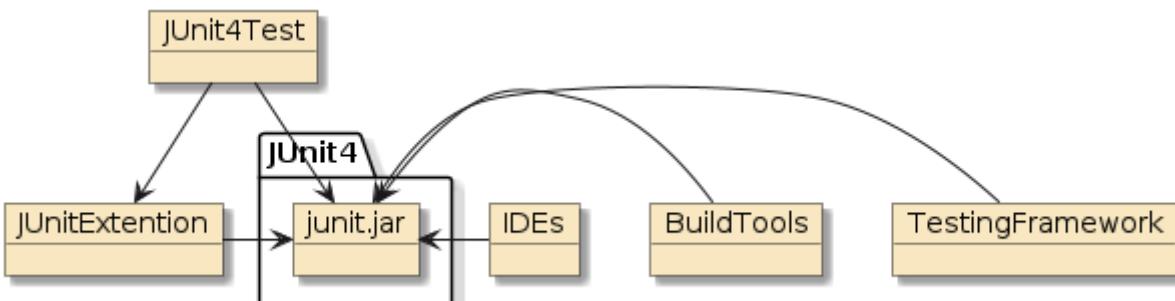


Figure 25. Everyone uses `junit.jar`

107.2. JUnit 5 Areas

The next iteration of JUnit involved a total rewrite — that separated the overall project into three (3) modules.

- JUnit Platform
 - foundation for launching tests on a JVM — from anything
 - defines **TestEngine** API for JUnit and **3rd party TestEngines and Extensions** to use
- JUnit Jupiter ("new stuff")
 - evolution from legacy
 - provides **TestEngine** for running Jupiter-based tests
- JUnit Vintage ("legacy stuff")
 - provides **TestEngine** for running JUnit 3 and JUnit 4-based tests

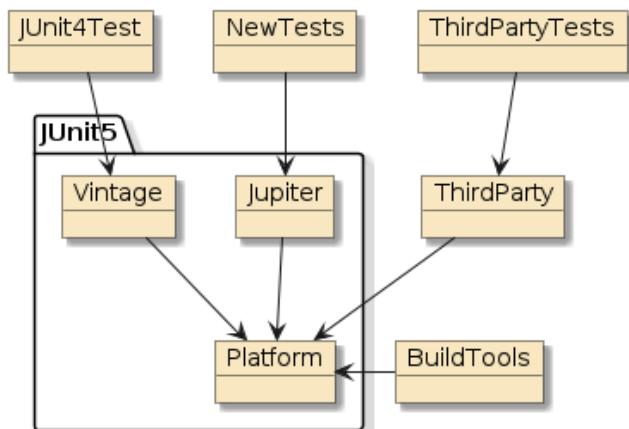


Figure 26. JUnit 5 Modularization



The name Jupiter was selected because it is the 5th planet from the Sun

107.3. JUnit 5 Module JARs

The JUnit 5 modules have several JARs within them that separate interface from implementation — ultimately decoupling the test code from the core engine.

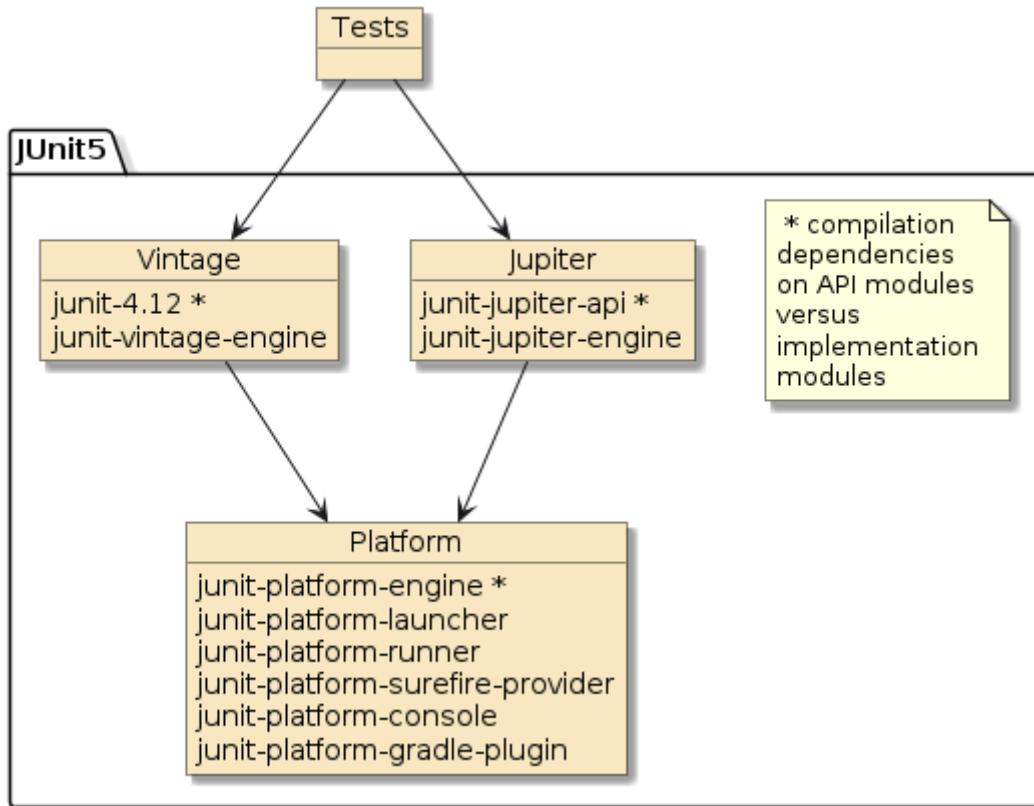


Figure 27. JUnit 5 [Module Contents](#)

Chapter 108. Syntax Basics

Before getting too deep into testing, I think it is a good idea to make a very shallow pass at the technical stack we will be leveraging.

- JUnit
- Mockito
- Spring Boot

Each of the example tests that follow can be run within the IDE at the method, class, and parent java package level. The specifics of each IDE will not be addressed here but I will cover some Maven details once we have a few tests defined.

Chapter 109. JUnit Vintage Basics

It is highly likely that projects will have JUnit 4-based tests around for a significant amount of time without good reason to update them—because we do not have to. There is full backwards-compatibility support within JUnit 5 and the specific libraries to enable that are automatically included by [spring-boot-starter-test](#). The following example shows a basic JUnit example using the Vintage syntax.

109.1. JUnit Vintage Example Lifecycle Methods

Basic JUnit Vintage Example Lifecycle Methods

```
package info.ejava.examples.app.testing.testbasics.vintage;

import lombok.extern.slf4j.Slf4j;
import org.junit.*;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;

@Slf4j
public class ExampleUnit4Test {
    @BeforeClass
    public static void setUpClass() {
        log.info("setUpClass");
    }
    @Before
    public void setUp() {
        log.info("setUp");
    }
    @After
    public void tearDown() {
        log.info("tearDown");
    }
    @AfterClass
    public static void tearDownClass() {
        log.info("tearDownClass");
    }
}
```

- annotations come from the [org.junit.*](#) Java package
- lifecycle annotations are
 - `@BeforeClass`—a public static method run before the first `@Before` method call and all tests within the class
 - `@Before` - a public instance method run before each test in the class
 - `@After` - a public instance method run after each test in the class
 - `@AfterClass` - a public static method run after all tests within the class and the last `@After`

method called

109.2. JUnit Vintage Example Test Methods

Basic JUnit Vintage Example Test Methods

```
@Test(expected = IllegalArgumentException.class)
public void two_plus_two() {
    log.info("2+2=4");
    assertEquals(4, 2+2);
    throw new IllegalArgumentException("just demonstrating expected exception");
}
@Test
public void one_and_one() {
    log.info("1+1=2");
    assertTrue("problem with 1+1", 1+1==2);
    assertEquals(String.format("problem with %d+%d", 1, 1), 2, 1+1);
}
```

- @Test - a public instance method where subjects are invoked and result assertions are made
- exceptions can be asserted at overall method level—but not at a specific point in the method and exception itself cannot be inspected without switching to a manual try/catch technique
- asserts can be augmented with a String message in the first position
 - the expense of building String message is always paid whether needed or not

```
assertEquals(String.format("problem with %d+%d", 1, 1), 2, 1+1);
```



Vintage requires the class and methods have public access.

109.3. JUnit Vintage Basic Syntax Example Output

The following example output shows the lifecycle of the setup and teardown methods combined with two test methods. Note that:

- the static @BeforeClass and @AfterClass methods are run once
- the instance @Before and @After methods are run for each test

Basic JUnit Vintage Example Output

```
16:35:42.293 INFO ...testing.testbasics.vintage.ExampleJUnit4Test - setUpClass ①
16:35:42.297 INFO ...testing.testbasics.vintage.ExampleJUnit4Test - setUp ②
16:35:42.297 INFO ...testing.testbasics.vintage.ExampleJUnit4Test - 2+2=4
16:35:42.297 INFO ...testing.testbasics.vintage.ExampleJUnit4Test - tearDown ②
16:35:42.299 INFO ...testing.testbasics.vintage.ExampleJUnit4Test - setUp ②
16:35:42.300 INFO ...testing.testbasics.vintage.ExampleJUnit4Test - 1+1=2
16:35:42.300 INFO ...testing.testbasics.vintage.ExampleJUnit4Test - tearDown ②
16:35:42.300 INFO ...testing.testbasics.vintage.ExampleJUnit4Test - tearDownClass ①
```

① @BeforeClass and @AfterClass called once per test class

② @Before and @After executed for each @Test



Not demonstrated—a new instance of the test class is instantiated for each test. No object state is retained from test to test without the manual use of static variables.



JUnit Vintage provides no construct to dictate repeatable ordering of test methods within a class—thus making it hard to use test cases to depict lengthy, deterministically ordered scenarios.

Chapter 110. JUnit Jupiter Basics

To simply change-over from Vintage to Jupiter syntax, there are a few minor changes.

- annotations and assertions have changed packages from `org.junit` to `org.junit.jupiter.api`
- lifecycle annotations have changed names
- assertions have changed the order of optional arguments
- exceptions can now be explicitly tested and inspected within the test method body



Vintage no longer requires classes or methods to be public. Anything non-private should work.

110.1. JUnit Jupiter Example Lifecycle Methods

The following example shows a basic JUnit example using the Jupiter syntax.

Basic JUnit Jupiter Example Lifecycle Methods

```
package info.ejava.examples.app.testing.testbasics.jupiter;

import lombok.extern.slf4j.Slf4j;
import org.junit.jupiter.api.*;

import static org.junit.jupiter.api.Assertions.*;

@Slf4j
class ExampleJUnit5Test {
    @BeforeAll
    static void setUpClass() {
        log.info("setUpClass");
    }
    @BeforeEach
    void setUp() {
        log.info("setUp");
    }
    @AfterEach
    void tearDown() {
        log.info("tearDown");
    }
    @AfterAll
    static void tearDownClass() {
        log.info("tearDownClass");
    }
}
```

- annotations come from the `org.junit.jupiter.*` Java package
- lifecycle annotations are

- @BeforeAll—a static method run before the first @BeforeEach method call and all tests within the class
- @BeforeEach - an instance method run before each test in the class
- @AfterEach - an instance method run after each test in the class
- @AfterAll - a static method run after all tests within the class and the last @AfterEach method called

110.2. JUnit Jupiter Example Test Methods

Basic JUnit Jupiter Example Test Methods

```

@Test
void two_plus_two() {
    log.info("2+2=4");
    assertEquals(4, 2+2);
    Exception ex=assertThrows(IllegalArgumentException.class, () ->{
        throw new IllegalArgumentException("just demonstrating expected exception");
    });
    assertTrue(ex.getMessage().startsWith("just demo"));
}

@Test
void one_and_one() {
    log.info("1+1=2");
    assertTrue(1+1==2, "problem with 1+1");
    assertEquals(2, 1+1, ()->String.format("problem with %d+%d",1,1));
}

```

- @Test - a instance method where assertions are made
- exceptions can now be explicitly tested at a specific point in the test method—permitting details of the exception to also be inspected
- asserts can be augmented with a String message in the last position
 - this is a breaking change from Vintage syntax
 - the expense of building complex String messages can be deferred to a lambda function

assertEquals(2, 1+1, ()->String.format("problem with %d+%d",1,1));

110.3. JUnit Jupiter Basic Syntax Example Output

The following example output shows the lifecycle of the setup/teardown methods combined with two test methods. The default logger formatting added the new lines in between tests.

Basic JUnit Jupiter Example Output

```

16:53:44.852 INFO ...testing.testbasics.jupiter.ExampleJUnit5Test - setUpClass ①
③
16:53:44.866 INFO ...testing.testbasics.jupiter.ExampleJUnit5Test - setUp ②

```

```
16:53:44.869 INFO ...testing.testbasics.jupiter.ExampleJUnit5Test - 2+2=4
16:53:44.874 INFO ...testing.testbasics.jupiter.ExampleJUnit5Test - tearDown ②
③

16:53:44.879 INFO ...testing.testbasics.jupiter.ExampleJUnit5Test - setUp ②
16:53:44.880 INFO ...testing.testbasics.jupiter.ExampleJUnit5Test - 1+1=2
16:53:44.881 INFO ...testing.testbasics.jupiter.ExampleJUnit5Test - tearDown ②
③
16:53:44.883 INFO ...testing.testbasics.jupiter.ExampleJUnit5Test - tearDownClass ①
```

① @BeforeAll and @AfterAll called once per test class

② @Before and @After executed for each @Test

③ The default IDE logger formatting added the new lines in between tests



Not demonstrated—we have the default [option to have a new instance per test](#) like [Vintage](#) or [same instance for all tests](#) and a [defined test method order](#)—which allows for lengthy scenario tests to be broken into increments. See [@TestInstance](#) annotation and [TestInstance.Lifecycle](#) enum for details.

Chapter 111. JUnit Jupiter Test Case Adjustments

111.1. Test Instance

State used by tests can be expensive to create or outside the scope of individual tests. JUnit allows this state to be initialized and shared between test methods using one of two test instance techniques using the `@TestInstance` annotation.

111.1.1. Shared Static State - PER_METHOD

The default test instance is `PER_METHOD`. With this option, the instance of the class is torn down and re-instantiated between each test. We must declare any shared state as `static` to have it live during the lifecycle of all instance methods. The `@BeforeAll` and `@AfterAll` methods that initialize and tear down this data must be declared static when using `PER_METHOD`.

TestInstance.PER_METHOD Shared State Example

```
@TestInstance(TestInstance.Lifecycle.PER_METHOD) //the default ①
class StaticShared {
    private static int staticState; ②
    @BeforeAll
    static void init() { ③
        log.info("state={}", staticState++);
    }
    @Test
    void testA() { log.info("state={}", staticState); } ④
    @Test
    void testB() { log.info("state={}", staticState); }
```

① test case class is instantiated per method

② any shared state must be declared private

③ `@BeforeAll` and `@AfterAll` methods must be declared static

④ `@Test` methods are normal instance methods with access to the static state

111.2. Shared Instance State - PER_CLASS

There are often times during an integration test where shared state (e.g., injected components) is only available once the test case is instantiated. We can make instance state sharable by using the `PER_CLASS` option. This makes the test case injectable by the container.

TestInstance.PER_CLASS Shared State Example

```
@TestInstance(TestInstance.Lifecycle.PER_CLASS) ①
class InstanceShared {
    private int instanceState; ②
```

```

@BeforeAll
void init() { ③
    log.info("state={}", instanceState++);
}
@Test
void testA() { log.info("state={}", instanceState); }
@Test
void testB() { log.info("state={}", instanceState); }

```

- ① one instance is created for all tests
- ② any shared state must be declared private
- ③ `@BeforeAll` and `@AfterAll` methods must be declared **non-static**



Use of `@ParameterizedTest` and `@MethodSource` (later topics), where the method source uses components injected into the test case instance is one example of when one needs to use `PER_CLASS`.

111.2.1. Test Ordering

Although it is a "best practice" to make tests independent and be executed in any order — there can be times when one wants a specified order. There are a few options: ^[1]

- Random Order
- Specified Order
- by Method Name
- by Display Name
- (custom order)

Method Ordering Options

```

...
import org.junit.jupiter.api.*;

@TestMethodOrder(
//      MethodOrderer.OrderAnnotation.class
//      MethodOrderer.MethodName.class
//      MethodOrderer.DisplayName.class
      MethodOrderer.Random.class
)
class ExampleJUnit5Test {
    @Test
    @Order(1)
    void two_plus_two() {
        ...
    }

    @Test
    @Order(2)
    void one_and_one() {
        ...
    }
}

```

Explicit Method Ordering is the Exception



It is best practice to make test cases and tests within test cases modular and independent of one another. To require a specific order violates that practice—but sometimes there are reasons to do so. One example violation is when the overall test case is broken down into test methods that addresses a multi-step scenario. In older versions of JUnit—that would have been required to be a single `@Test` calling out to helper methods.

[1] "[The Order of Tests in JUnit](#)", Baeldung, May 2022

Chapter 112. Assertion Basics

The setup methods (`@BeforeAll` and `@BeforeEach`) of the test case and early parts of the test method (`@Test`) allow for us to define a given test context and scenario for the subject of the test. Assertions are added to the evaluation portion of the test method to determine whether the subject performed correctly. The result of the assertions determine the pass/fail of the test.

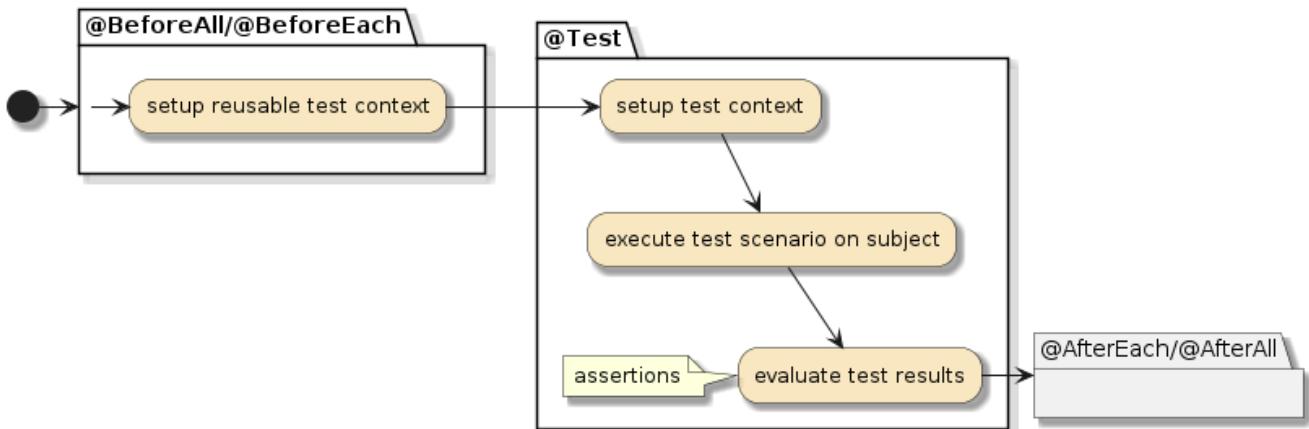


Figure 28. Assertions are Key Point in Tests

112.1. Assertion Libraries

There are three to four primary general purpose assertion libraries available for us to use within the `spring-boot-starter-test` suite before we start considering data format assertions for XML and JSON or add custom libraries of our own:

- JUnit - has built-in, basic assertions like True, False, Equals, NotEquals, etc.
 - Vintage - original assertions
 - Jupiter - same basic assertions with some new options and parameters swapped
- Hamcrest - uses natural-language [expressions for matches](#)
- AssertJ - an improvement to natural-language assertion expressions using type-based builders

The built-in JUnit assertions are functional enough to get any job done. The value in using the other libraries is their ability to express the assertion using natural-language terms without using a lot of extra, generic Java code.

112.1.1. JUnit Assertions

The assertions built into JUnit are basic and easy to understand — but limited in their expression. They have the basic form of taking subject argument(s) and the name of the static method is the assertion made about the arguments.

Example JUnit Assertion

```
import static org.junit.jupiter.api.Assertions.*;  
...
```

```
assertEquals(expected, lhs+rhs); ①
```

- ① JUnit static method assertions express assertion of one to two arguments

We are limited by the number of static assertion methods present and have to extend them by using code to manipulate the arguments (e.g., to be equal or true/false). However, once we get to that point—we can easily bring in robust assertion libraries. In fact, that is exactly what JUnit describes for us to do in the [JUnit User Guide](#).

112.1.2. Hamcrest Assertions

Hamcrest has a common pattern of taking a subject argument and a **Matcher** argument.

Example Hamcrest Assertion

```
import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.Matchers.equalTo;

...
	assertThat(beaver.getFirstName(), equalTo("Jerry")); ①
```

- ① LHS argument is value being tested, RHS `equalTo` returns an object implementing **Matcher** interface

The **Matcher** interface can be implemented by an unlimited number of expressions to implement the details of the assertion.

112.1.3. AssertJ Assertions

AssertJ uses a builder pattern that starts with the subject and then offers a nested number of assertion builders that are based on the previous node type.

Example AssertJ Assertion

```
import static org.assertj.core.api.Assertions.assertThat;
...
	assertThat(beaver.getFirstName()).isEqualTo("Jerry"); ①
```

- ① `assertThat` is a builder of assertion factories and `isEqual` executes an assertion in chain

Custom AssertJ Assertions

[Custom extensions](#) are accomplished by creating a new builder factory at the start of the call tree. See the following [link](#) for a small example.

AssertJ also provides an [Assertion Generator](#) that generates assertion source code based on specific POJO classes and templates we can override using a [maven](#) or [gradle](#) plugin. This allows us to express assertions about a **Person** class using the following syntax.

```
import static info.ejava.examples.app.testing.testbasics.Assertions.*;
...
assertThat(beaver).hasFirstName("Jerry");
```



IDEs have an easier time suggesting assertion builders with AssertJ because everything is a method call on the previous type. IDEs have a harder time suggesting Hamcrest matchers because there is very little to base the context on.

AssertJ Generator and Jakarta

Even though the Assertj [core library](#) has kept up to date, the [assertions generator plugin](#) has not. Current default execution of the plugin results in classes annotated with a `javax.annotation.Generated` annotation that has since been changed to `jakarta`.

I won't go into the details here, but the [class example](#) in gitlab shows where I downloaded the source templates from the [plugin source repository](#) and [edited](#) for use with Spring Boot 3 and Jakarta-based libraries.



A reply to one of the Assertj [support tickets](#) indicates they are working on it as a part of a Java 17 upgrade.

112.2. Example Library Assertions

The following example shows a small peek at the syntax for each of the four assertion libraries used within a JUnit Jupiter test case. They are shown without an `import static` declaration to better see where each comes from.

Example Assertions

```
package info.ejava.examples.app.testing.testbasics.jupiter;

import lombok.extern.slf4j.Slf4j;
import org.hamcrest.MatcherAssert;
import org.hamcrest.Matchers;
import org.junit.Assert;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;

@Slf4j
class AssertionsTest {
    int lhs=1;
    int rhs=1;
    int expected=2;

    @Test
    void one_and_one() {
```

```

//junit 4/Vintage assertion
Assert.assertEquals(expected, lhs+rhs); ①
//Jupiter assertion
Assertions.assertEquals(expected, lhs+rhs); ①
//hamcrest assertion
MatcherAssert.assertThat(lhs+rhs, Matchers.is(expected)); ②
//AssertJ assertion
org.assertj.core.api.Assertions.assertThat(lhs+rhs).isEqualTo(expected); ③
}
}

```

① JUnit assertions are expressed using a static method and one or more subject arguments

② Hamcrest asserts that the subject matches a `Matcher` that can be infinitely extended

③ AssertJ's extensible subject assertion provides type-specific assertion builders

112.3. Assertion Failures

Assertions will report a generic message when they fail. If we change the expected result of the example from 2 to 3, the following error message will be reported. It contains a generic message of the assertion failure (location not shown) without context other than the test case and test method it was generated from (not shown).

Example Default Assert Failure Message

```
java.lang.AssertionError: expected:<3> but was:<2> ①
```

① we are not told what 3 and 2 are within a test except that 3 was expected and they are not equal

112.3.1. Adding Assertion Context

However, there are times when some additional text can help to provide more context about the problem. The following example shows the previous test augmented with an optional message. Note that JUnit Jupiter assertions permit the lazy instantiation of complex message strings using a lambda. AssertJ provides for lazy instantiation using `String.format` built into the `as()` method.

Example Assertions with Message Supplied

```

@Test
void one_and_one_description() {
    //junit 4/Vintage assertion
    Assert.assertEquals("math error", expected, lhs+rhs); ①
    //Jupiter assertions
    Assertions.assertEquals(expected, lhs+rhs, "math error"); ②
    Assertions.assertEquals(expected, lhs+rhs,
        ()->String.format("math error %d+%d!=%d",lhs,rhs,expected)); ③
    //hamcrest assertion
    MatcherAssert.assertThat("math error",lhs+rhs, Matchers.is(expected)); ④
    //AssertJ assertion
    org.assertj.core.api.Assertions.assertThat(lhs+rhs)
}

```

```

        .as("math error") ⑤
        .isEqualTo(expected);
org.assertj.core.api.Assertions.assertThat(lhs+rhs)
        .as("math error %d+%d!=%d",lhs, rhs, expected) ⑥
        .isEqualTo(expected);
}

```

- ① JUnit Vintage syntax places optional message as first parameter
- ② JUnit Jupiter moves the optional message to the last parameter
- ③ JUnit Jupiter also allows optional message to be expressed thru a lambda function
- ④ Hamcrest passes message in first position like JUnit Vintage syntax
- ⑤ AspectJ uses an `as()` builder method to supply a message
- ⑥ AspectJ also supports `String.format` and args when expressing message

Example Assert Failure with Supplied Message

```
java.lang.AssertionError: math error expected:<3> but was:<2> ①
```

- ① an extra "math error" was added to the reported error to help provide context



Although AssertJ supports multiple asserts in a single call chain, your description (`as("description")`) must come before the first failing assertion.

Because AssertJ uses chaining



- there are fewer imports required
- IDEs are able to more easily suggest a matcher based on the type returned from the end of the chain. There is always a context specific to the next step.

112.4. Testing Multiple Assertions

The above examples showed several ways to assert the same thing with different libraries. However, evaluation would have stopped at the first failure in each test method. There are many times when we want to know the results of several assertions. For example, take the case where we are testing different fields in a returned object (e.g., `person.getFirstName()`, `person.getLastName()`). We may want to see all the results to give us better insight for the entire problem.

JUnit Jupiter and AssertJ support testing multiple assertions prior to failing a specific test and then go on to report the results of each failed assertion.

112.4.1. JUnit Jupiter Multiple Assertion Support

JUnit Jupiter uses a variable argument list of Java 8 lambda functions in order to provide support for testing multiple assertions prior to failing a test. The following example will execute both assertions and report the result of both when they fail.

```

    @Test
    void junit_all() {
        Assertions.assertAll("all assertions",
            () -> Assertions.assertEquals(expected, lhs+rhs, "jupiter assertion"), ①
            () -> Assertions.assertEquals(expected, lhs+rhs,
                ()->String.format("jupiter format %d+%d!=%d",lhs,rhs,expected))
        );
    }
}

```

① JUnit Jupiter uses Java 8 lambda functions to execute and report results for multiple assertions

112.4.2. AssertJ Multiple Assertion Support

AssertJ uses a special factory class ([SoftAssertions](#)) to build assertions from to support that capability. Notice also that we have the chance to inspect the state of the assertions before failing the test. That can give us the chance to gather additional information to place into the log. We also have the option of not technically failing the test under certain conditions.

AssertJ Multiple Assertion Support

```

import org.assertj.core.api.SoftAssertions;
...
    @Test
    public void all() {
        Person p = beaver; //change to eddie to cause failures
        SoftAssertions softly = new SoftAssertions(); ①
        softly.assertThat(p.getFirstName()).isEqualTo("Jerry");
        softly.assertThat(p.getLastName()).isEqualTo("Mathers");
        softly.assertThat(p.getDob()).isAfter(wally.getDob());

        log.info("error count={}", softly.errorsCollected().size()); ②
        softly.assertAll(); ③
    }
}

```

① a special [SoftAssertions](#) builder is used to construct assertions

② we are able to inspect the status of the assertions before failure thrown

③ assertion failure thrown during later `assertAll()` call

112.5. Asserting Exceptions

JUnit Jupiter and AssertJ provide direct support for inspecting Exceptions within the body of the test method. Surprisingly, Hamcrest offers no built-in matchers to directly inspect Exceptions.

112.5.1. JUnit Jupiter Exception Handling Support

JUnit Jupiter allows for an explicit testing for Exceptions at specific points within the test method.

The type of Exception is checked and made available to follow-on assertions to inspect. From that point forward JUnit assertions do not provide any direct support to inspect the Exception.

JUnit Jupiter Exception Handling Support

```
import org.junit.jupiter.api.Assertions;  
...  
@Test  
public void exceptions() {  
    RuntimeException ex1 = Assertions.assertThrows(RuntimeException.class, ①  
        () -> {  
            throw new IllegalArgumentException("example exception");  
        });  
}
```

① JUnit Jupiter provides means to assert an Exception thrown and provide it for inspection

112.5.2. AssertJ Exception Handling Support

AssertJ has an Exception testing capability that is similar to JUnit Jupiter—where an explicit check for the Exception to be thrown is performed and the thrown Exception is made available for inspection. The big difference here is that AssertJ provides Exception assertions that can directly inspect the properties of Exceptions using natural-language calls.

AssertJ Exception Handling and Inspection Support

```
Throwable ex1 = catchThrowable( ①  
    ()->{ throw new IllegalArgumentException("example exception"); });  
assertThat(ex1).hasMessage("example exception"); ②  
  
RuntimeException ex2 = catchThrowableOfType( ①  
    ()->{ throw new IllegalArgumentException("example exception"); },  
    RuntimeException.class);  
assertThat(ex1).hasMessage("example exception"); ②
```

① AssertJ provides means to assert an Exception thrown and provide it for inspection

② AssertJ provides assertions to directly inspect Exceptions

AssertJ goes one step further by providing an assertion that not only is the exception thrown, but can also tack on assertion builders to make on-the-spot assertions about the exception thrown. This has the same end functionality as the previous example—except:

- previous method returned the exception thrown that can be subject to independent inspection
- this technique returns an assertion builder with the capability to build further assertions against the exception

AssertJ Integrated Exception Handling Support

```
assertThatThrownBy( ①  
    () -> {
```

```

        throw new IllegalArgumentException("example exception");
    }).hasMessage("example exception");

assertThatExceptionOfType(RuntimeException.class).isThrownBy( ①
    () -> {
        throw new IllegalArgumentException("example exception");
    }).withMessage("example exception");

```

- ① AssertJ provides means to use the caught Exception as an assertion factory to directly inspect the Exception in a single chained call

112.6. Asserting Dates

AssertJ has built-in support for date assertions. We have to add a separate library to gain date matchers for Hamcrest.

112.6.1. AssertJ Date Handling Support

The following shows an example of AssertJ's built-in, natural-language support for Dates.

AssertJ Exception Handling Support

```

import static org.assertj.core.api.Assertions.*;
...
@Test
public void dateTypes() {
    assertThat(beaver.getDob()).isAfter(wally.getDob());
    assertThat(beaver.getDob())
        .as("beaver NOT younger than wally")
        .isAfter(wally.getDob()); ①
}

```

- ① AssertJ builds date assertions that directly inspect dates using natural-language

112.6.2. Hamcrest Date Handling Support

Hamcrest can be extended to support date matches by adding an external `hamcrest-date` library.

Hamcrest Date Support Dependency

```

<!-- for hamcrest date comparisons -->
<dependency>
    <groupId>org.exparity</groupId>
    <artifactId>hamcrest-date</artifactId>
    <version>2.0.7</version>
    <scope>test</scope>
</dependency>

```

That dependency adds at least a `DateMatchers` class with date matchers that can be used to express

date assertions using natural-language expression.

Hamcrest Date Handling Support

```
import org.exparity.hamcrest.date.DateMatchers;
import static org.hamcrest.MatcherAssert.assertThat;
...
@Test
public void dateTypes() {
    //requires additional org.exparity:hamcrest-date library
    assertThat(beaver.getDob(), DateMatchers.after(wally.getDob()));
    assertThat("beaver NOT younger than wally", beaver.getDob(),
               DateMatchers.after(wally.getDob())); ①
}
```

① `hamcrest-date` adds matchers that can directly inspect dates

Chapter 113. Mockito Basics

Without much question—we will have more complex software to test than what we have briefly shown so far in this lesson. The software will inevitably be structured into layered dependencies where one layer cannot be tested without the lower layers it calls. To implement unit tests, we have a few choices:

1. use the real lower-level components (i.e., "all the way to the DB and back", remember—I am calling that choice "Unit Integration Tests" if it can be technically implemented/managed within a single JVM)
2. create a stand-in for the lower-level components (aka "test double")

We will likely take the first approach during integration testing but the lower-level components may bring in too many dependencies to realistically test during a separate unit's own detailed testing.

113.1. Test Doubles

The second approach ("test double") has a [few options](#):

- fake - using a scaled down version of the real component (e.g., in-memory SQL database)
- stub - simulation of the real component by using pre-cached test data
- mock - defining responses to calls and the ability to inspect the actual incoming calls made

113.2. Mock Support

`spring-boot-starter-test` brings in a pre-integrated, mature [open source mocking framework](#) called Mockito. See the example below for an example unit test augmented with mocks using Mockito. It uses a simple Java `Map<String, String>` to demonstrate some simulation and inspection concepts. In a real unit test, the Java Map interface would stand for:

- an interface we are designing (i.e., [testing the interface contract we are designing from the client-side](#))
- a test double we want to inject into a component under test that will answer with pre-configured answers and be able to inspect how called (e.g., testing collaborations within a [white box](#) test)

113.3. Mockito Learning Example Declarations

Basic Mockito Example Declarations

```
package info.ejava.examples.app.testing.testbasics.mockito;

import org.junit.jupiter.api.*;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.ArgumentCaptor;
```

```

import org.mockito.Captor;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;

import java.util.Map;

import static org.assertj.core.api.Assertions.assertThat;
import static org.mockito.Mockito.*;

@ExtendWith(MockitoExtension.class)
public class ExampleMockitoTest {
    @Mock //creating a mock to configure for use in each test
    private Map<String, String> mapMock;
    @Captor
    private ArgumentCaptor<String> stringArgCaptor;

```

- `@ExtendWith` bootstraps Mockito behavior into test case
- `@Mock` can be used to inject a mock of the defined type
 - "nice mock" is immediately available - will react in potentially useful manner by default
- `@Captor` can be used to capture input parameters passed to the mock calls



`@InjectMocks` will be demonstrated in later white box testing — where the defined mocks get injected into component under test.

113.4. Mockito Learning Example Test



The following learning example provides a demonstration of Mock capability — using only the Mock, without the component under test. This is not something anyone would do because it is only demonstrating the Mock capability versus testing the component under test using the assembled Mock. The calls to the Mock during the "conduct test" are calls we would anticipate the component under test would make while being tested.

Basic Mockito Learning Example Test

```

@Test
public void listMap() {
    //define behavior of mock during test
    when(mapMock.get(stringArgCaptor.capture()))
        .thenReturn("springboot", "testing"); ①

    //conduct test
    int size = mapMock.size();
    String secret1 = mapMock.get("happiness");
    String secret2 = mapMock.get("joy");

    //evaluate results

```

```
verify(mapMock).size(); //verify called once ③
verify(mapMock, times(2)).get(anyString()); //verify called twice
//verify what was given to mock
assertThat(stringArgCaptor.getAllValues().get(0)).isEqualTo("happiness"); ②
assertThat(stringArgCaptor.getAllValues().get(1)).isEqualTo("joy");
//verify what was returned by mock
assertThat(size).as("unexpected size").isEqualTo(0);
assertThat(secret1).as("unexpected first result").isEqualTo("springboot");
assertThat(secret2).as("unexpected second result").isEqualTo("testing");
}
```

- ① when()/then() define custom conditions and responses for mock within scope of test
- ② getValue()/getAllValues() can be called on the captor to obtain value(s) passed to the mock
- ③ verify() can be called to verify what was called of the mock



`mapMock.size()` returned 0 while `mapMock.get()` returned values. We defined behavior for `mapMock.get()` but left other interface methods in their default, "nice mock" state.

Chapter 114. BDD Acceptance Test Terminology

Behavior-Driven Development (BDD) can be part of an agile development process and adds the use of natural-language constructs to express behaviors and outcomes. The BDD [behavior specifications](#) are stories with a certain structure that contain an acceptance criteria that follows a "given", "when", "then" structure:

- **given** - initial context
- **when** - event triggering scenario under test
- **then** - expected outcome

114.1. Alternate BDD Syntax Support

There is also a strong push to express acceptance criteria in code that can be executed versus a document. Although far from a perfect solution, JUnit, AssertJ, and Mockito do provide some syntax support for BDD-based testing:

- **JUnit Jupiter** allows the assignment of meaningful natural-language phrases for test case and test method names. Nested classes can also be employed to provide additional expression.
- **Mockito** defines alternate method names to better map to the given/when/then language of BDD
- **AssertJ** defines alternate assertion factory names using `then()` and `and.then()` wording

114.2. Example BDD Syntax Support

The following shows an example use of the BDD syntax.

Example BDD Syntax Support

```
import org.junit.jupiter.api.*;
import static org.assertj.core.api.BDDAssertions.and;
import static org.mockito.BDDMockito.given;
import static org.mockito.BDDMockito.then;

@ExtendWith(MockitoExtension.class)
@DisplayNameGeneration(DisplayNameGenerator.ReplaceUnderscores.class) ①
@DisplayName("map") ②
public class ExampleMockitoTest {

    ...
    @Nested ③
    public class when_has_key { ①
        @Test
        public void returns_values() {
            //given
            given(mapMock.get(stringArgCaptor.capture()))
        }
    }
}
```

```

        .willReturn("springboot", "testing"); ④
//alt syntax
//      doReturn("springboot", "testing")
//          .when(mapMock).get(stringArgCaptor.capture());
//when
int size = mapMock.size();
String secret1 = mapMock.get("happiness");
String secret2 = mapMock.get("joy");

//then - can use static import for BDDMockito or BDDAssertions, not both
then(mapMock).should().size(); //verify called once ⑤
then(mapMock).should(times(2)).get(anyString()); //verify called twice
⑦ ⑥
and.then(stringArgCaptor.getAllValues().get(0)).isEqualTo("happiness");
and.then(stringArgCaptor.getAllValues().get(1)).isEqualTo("joy");
and.then(size).as("unexpected size").isEqualTo(0);
and.then(secret1).as("unexpected first result").isEqualTo("springboot");
and.then(secret2).as("unexpected second result").isEqualTo("testing");
}
}

```

- ① JUnit `DisplayNameGenerator.ReplaceUnderscores` will form a natural-language display name by replacing underscores with spaces
- ② JUnit `DisplayName` sets the display name to a specific value
- ③ JUnit `Nested` classes can be used to better express test context
- ④ Mockito `when/then` syntax replaced by `given/will` syntax expresses the definition of the mock
- ⑤ Mockito `verify/then` syntax replaced by `then/should` syntax expresses assertions made on the mock
- ⑥ AssertJ `then` syntax expresses assertions made to supported object types
- ⑦ AssertJ `and` field provides a natural-language way to access both AssertJ `then` and Mockito `then` in the same class/method



AssertJ provides a static final `and` field to allow its static `then()` and Mockito's static `then()` to be accessed in the same class/test

114.3. Example BDD Syntax Output

When we run our test—the following natural-language text is displayed.

▼	✓ Test Results	793 ms
▼	✓ map	793 ms
▼	✓ when has key	793 ms
	✓ returns values	793 ms

Figure 29. Example BDD Syntax Output

114.4. JUnit Options Expressed in Properties

We can define a global setting for the display name generator using `junit-platform.properties`

test-classes/junit-platform.properties

```
junit.jupiter.displayname.generator.default = \
    org.junit.jupiter.api.DisplayNameGenerator$ReplaceUnderscores
```

This can also be used to [express](#):

- method order
- class order
- test instance lifecycle
- `@Parameterized` test naming
- parallel execution

Chapter 115. Tipping Example

To go much further describing testing — we need to assemble a small set of interfaces and classes to test. I am going to use a common problem when several people go out for a meal together and need to split the check after factoring in the tip.

- **TipCalculator** - returns the amount of tip required when given a certain bill total and rating of service. We could have multiple evaluators for tips and have defined an interface for clients to depend upon.
- **BillCalculator** - provides the ability to calculate the share of an equally split bill given a total, service quality, and number of people.

The following class diagram shows the relationship between the interfaces/classes. They will be the subject of the following Unit Integration Tests involving the Spring context.

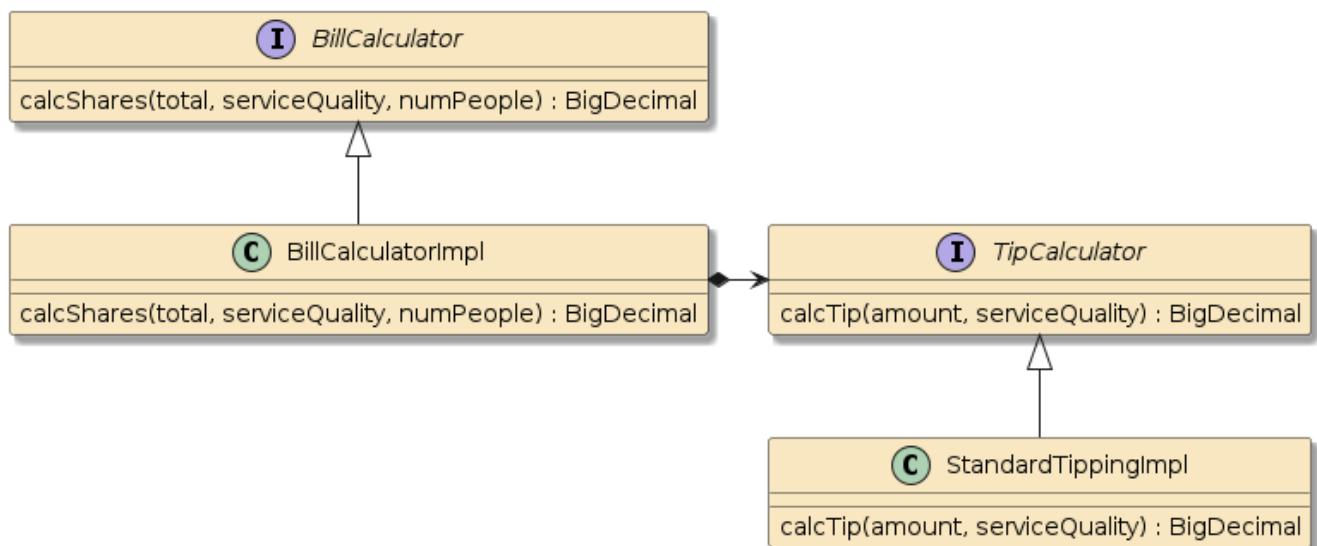


Figure 30. Tipping Example Class Model

Chapter 116. Review: Unit Test Basics

In previous chapters we have looked at pure unit test constructs with an eye on JUnit, assertion libraries, and a little of Mockito. In preparation for the unit integration topic and adding the Spring context in the following chapter—I want to review the simple test constructs in terms of the Tipping example.

116.1. Review: POJO Unit Test Setup

Review: POJO Unit Test Setup

```
@DisplayNameGeneration(DisplayNameGenerator.ReplaceUnderscores.class) ①
@GeneratedValue("Standard Tipping Calculator")
public class StandardTippingCalculatorImplTest {
    //subject under test
    private TipCalculator tipCalculator; ②

    @BeforeEach ③
    void setup() { //simulating a complex initialization
        tipCalculator=new StandardTippingImpl();
    }
}
```

① DisplayName is part of BDD naming and optional for all tests

② there will be one or more objects under test. These will be POJOs.

③ @BeforeEach plays the role of a the container—wiring up objects under test

116.2. Review: POJO Unit Test

The unit test is being expressed in terms of BDD conventions. It is broken up into "given", "when", and "then" blocks and highlighted with use of BDD syntax where provided (JUnit and AssertJ in this case).

Review: POJO Unit Test

```
@Test
public void given_fair_service() { ①
    //given - a $100 bill with FAIR service ②
    BigDecimal billTotal = new BigDecimal(100);
    ServiceQuality serviceQuality = ServiceQuality.FAIR;

    //when - calculating tip ②
    BigDecimal resultTip = tipCalculator.calcTip(billTotal, serviceQuality);

    //then - expect a result that is 15% of the $100 total ②
    BigDecimal expectedTip = billTotal.multiply(BigDecimal.valueOf(0.15));
    then(resultTip).isEqualTo(expectedTip); ③
}
```

- ① using JUnit snake_case natural language expression for test name
- ② BDD convention of given, when, then blocks. Helps to be short and focused
- ③ using AssertJ assertions with BDD syntax

116.3. Review: Mocked Unit Test Setup

The following example moves up a level in the hierarchy and forces us to test a class that had a dependency. A pure unit test would mock out all dependencies—which we are doing for [TipCalculator](#).

Review: Mocked Unit Test Setup

```
@ExtendWith(MockitoExtension.class) ①
@DisplayNameGeneration(DisplayNameGenerator.ReplaceUnderscores.class)
@DisplayName("Bill CalculatorImpl Mocked Unit Test")
public class BillCalculatorMockedTest {
    //subject under test
    private BillCalculator billCalculator;

    @Mock ②
    private TipCalculator tipCalculatorMock;

    @BeforeEach
    void init() { ③
        billCalculator = new BillCalculatorImpl(tipCalculatorMock);
    }
}
```

- ① Add Mockito extension to JUnit
- ② Identify which interfaces to Mock
- ③ In this example, we are manually wiring up the subject under test

116.4. Review: Mocked Unit Test

The following shows the TipCalculator mock being instructed on what to return based on input criteria and making call activity available to the test.

Review: Mocked Unit Test

```
@Test
public void calc_shares_for_people_including_tip() {
    //given - we have a bill for 4 people and tip calculator that returns tip amount
    BigDecimal billTotal = new BigDecimal(100.0);
    ServiceQuality service = ServiceQuality.GOOD;
    BigDecimal tip = billTotal.multiply(new BigDecimal(0.18));
    int numPeople = 4;
    //configure mock
    given(tipCalculatorMock.calcTip(billTotal, service)).willReturn(tip); ①
```

```

//when - call method under test
BigDecimal shareResult = billCalculator.calcShares(billTotal, service, numPeople);

//then - tip calculator should be called once to get result
then(tipCalculatorMock).should(times(1)).calcTip(billTotal,service); ②

//verify correct result
BigDecimal expectedShare = billTotal.add(tip).divide(new BigDecimal(numPeople));
and.then(shareResult).isEqualTo(expectedShare);
}

```

- ① configuring response behavior of Mock
- ② optionally inspecting subject calls made

116.5. @InjectMocks

The final unit test example shows how we can leverage Mockito to instantiate our subject(s) under test and inject them with mocks. That takes over at least one job the `@BeforeEach` was performing.

Alternative Mocked Unit Test

```

@ExtendWith(MockitoExtension.class)
@DisplayNameGeneration(DisplayNameGenerator.ReplaceUnderscores.class)
@DisplayName("Bill CalculatorImpl")
public class BillCalculatorImplTest {
    @Mock
    TipCalculator tipCalculatorMock;
    /*
    Mockito is instantiating this implementation class for us an injecting Mocks
    */
    @InjectMocks ①
    BillCalculatorImpl billCalculator;
}

```

- ① instantiates and injects out subject under test

Chapter 117. Spring Boot Unit Integration Test Basics

Pure unit testing can be efficiently executed without a Spring context, but there will eventually be a time to either:

- integrate peer components with one another (horizontal integration)
- integrate layered components to test the stack (vertical integration)

These goals are not easily accomplished without a Spring context and whatever is created outside of a Spring context will be different from production. Spring Boot and the Spring context can be brought into the test picture to more seamlessly integrate with other components and component infrastructure present in the end application. Although valuable, it will come at a performance cost and potentially add external resource dependencies—so don't look for it to replace the lightweight pure unit testing alternatives covered earlier.

117.1. Adding Spring Boot to Testing

There are two primary things that will change with our Spring Boot integration test:

1. define a Spring context for our test to operate using `@SpringBootTest`
2. inject components we wish to use/test from the Spring context into our tests using `@Autowired`



I found the following article: [Integration Tests with @SpringBootTest, by Tom Hombergs](#) and his "Testing with Spring Boot" series to be quite helpful in clarifying my thoughts and originally preparing these lecture notes. The [Spring Boot Testing](#) reference web page provides detailed coverage of the test constructs that go well beyond what I am covering at this point in the course. We will pick up more of that material as we get into web and data tier topics.

117.2. `@SpringBootTest`

To obtain a Spring context and leverage the auto-configuration capabilities of Spring Boot, we can take the easy way out and annotate our test with `@SpringBootTest`. This will instantiate a default Spring context based on the configuration defined or can be found.

`@SpringBootTest` Defines Spring Context for Test

```
package info.ejava.examples.app.testing.testbasics.tips;  
...  
import org.springframework.boot.test.context.SpringBootTest;  
...  
@SpringBootTest ①  
public class BillCalculatorNTest {
```

① using the default configuration search rules

117.3. Default @SpringBootConfiguration Class

By default, Spring Boot will look for a class annotated with `@SpringBootConfiguration` that is present at or above the Java package containing the test. Since we have a class in a parent directory that represents our `@SpringBootApplication` and that annotation wraps `@SpringBootConfiguration`, that class will be used to define the Spring context for our test.

Example @SpringBootConfiguration Class

```
package info.ejava.examples.app.testing.testbasics;  
...  
@SpringBootApplication  
// wraps => @SpringBootConfiguration  
public class TestBasicsApp {  
    public static void main(String...args) {  
        SpringApplication.run(TestBasicsApp.class,args);  
    }  
}
```

117.4. Conditional Components

When using the `@SpringBootApplication`, all components normally a part of the application will be part of the test. Be sure to define auto-configuration exclusions for any production components that would need to be turned off during testing.



```
@Configuration  
@ConditionalOnProperty(prefix="hello", name="enable", matchIfMissing=  
"true")  
public Hello quietHello() {  
    ...  
    @SpringBootTest(properties = { "hello.enable=false" }) ①
```

① test setting property to trigger disable of certain component(s)

117.5. Explicit Reference to @SpringBootConfiguration

Alternatively, we could have made an explicit reference as to which class to use if it was not in a standard relative directory or we wanted to use a custom version of the application for testing.

Explicit Reference to @SpringBootConfiguration Class

```
import info.ejava.examples.app.testing.testbasics.TestBasicsApp;  
...  
@SpringBootTest(classes = TestBasicsApp.class)  
public class BillCalculatorNTest {
```

117.6. Explicit Reference to Components

Assuming the components required for test is known and a manageable number...

Components Under Test

```
@Component  
@RequiredArgsConstructor  
public class BillCalculatorImpl implements BillCalculator {  
    private final TipCalculator tipCalculator;  
  
    ...  
  
    @Component  
    public class StandardTippingImpl implements TipCalculator {  
        ...
```

We can explicitly reference component classes needed to be in the Spring context.

Explicitly Referencing Components Under Test

```
@SpringBootTest(classes = {BillCalculatorImpl.class, StandardTippingImpl.class})  
public class BillCalculatorNTest {  
    @Autowired  
    BillCalculator billCalculator;
```

117.7. Active Profiles

Prior to adding the Spring context, Spring Boot configuration and logging conventions were not being enacted. However, now that we are bringing in a Spring context—we can designate special profiles to be activated for our context. This can allow us to define properties that are more relevant to our tests (e.g., expressive log context, increased log verbosity).

Example @ActiveProfiles Declaration

```
package info.ejava.examples.app.testing.testbasics.tips;  
  
import org.springframework.boot.test.context.SpringBootTest;  
import org.springframework.test.context.ActiveProfiles;  
  
@SpringBootTest  
@ActiveProfiles("test") ①  
public class BillCalculatorNTest {
```

① activating the "test" profile for this test

Example application-test.properties

```
# application-test.properties ①
```

```
logging.level.info.ejava.examples.app.testing.testbasics=DEBUG
```

- ① "test" profile setting loggers for package under test to **DEBUG** severity threshold

117.8. Example @SpringBootTest Unit Integration Test

Putting the pieces together, we have

- a complete Spring context
- **BillCalculator** injected into the test from the Spring context
- **TipCalculator** injected into billCalculator instance from Spring context
- a BDD natural-language, unit integration test that verifies result of bill calculator and tip calculator working together

```
@SpringBootTest
@ActiveProfiles("test")
@DisplayNameGeneration(DisplayNameGenerator.ReplaceUnderscores.class)
@DisplayName("bill calculator")
public class BillCalculatorNTest {
    @Autowired
    BillCalculator billCalculator;

    @Test
    public void calc_shares_for_bill_total() {
        //given
        BigDecimal billTotal = BigDecimal.valueOf(100.0);
        ServiceQuality service = ServiceQuality.GOOD;
        BigDecimal tip = billTotal.multiply(BigDecimal.valueOf(0.18));
        int numPeople = 4;

        //when - call method under test
        BigDecimal shareResult=billCalculator.calcShares(billTotal,service,numPeople);

        //then - verify correct result
        BigDecimal expectedShare = billTotal.add(tip).divide(BigDecimal.valueOf(4));
        then(shareResult).isEqualTo(expectedShare);
    }
}
```

117.9. Example @SpringBootTest NTest Output

When we run our test we get the following console information printed. Note that

- the **DEBUG** messages are from the **BillCalculatorImpl**
- **DEBUG** is being printed because the "test" profile is active and the "test" profile set the severity threshold for that package to be **DEBUG**

- method and line number information is also displayed because the test profile defines an expressive log event pattern

Example @SpringBootTest Unit Integration Test Output

```

  \\\ / ____'__--_(_)_--__-\_\_\_\_
(( )\___|'_|_|'_|_|'_\`|_\`|_\`|
 \\\ ___)|_|_|_|_|_|(|_|_| ) ) )
' |_____| .__|_|_|_|_|_\_,| / / / /
=====|_|=====|_|/_=/\_/_/_/
:: Spring Boot ::      (v3.3.2)

```

```

14:17:15.427 INFO BillCalculatorNTest#logStarting:55 - Starting BillCalculatorNTest
14:17:15.429 DEBUG BillCalculatorNTest#logStarting:56 - Running with Spring Boot
v2.2.6.RELEASE, Spring v5.2.5.RELEASE
14:17:15.430 INFO BillCalculatorNTest#logStartupProfileInfo:655 - The following
profiles are active: test
14:17:16.135 INFO BillCalculatorNTest#logStarted:61 - Started BillCalculatorNTest
in 6.155 seconds (JVM running for 8.085)
14:17:16.138 DEBUG BillCalculatorImpl#calcShares:24 - tip=$9.00, for $50.00 and
GOOD service
14:17:16.142 DEBUG BillCalculatorImpl#calcShares:33 - share=$14.75 for $50.00, 4
people and GOOD service
14:17:16.143 INFO BillHandler#run:24 - bill total $50.00, share=$14.75 for 4 people,
after adding tip for GOOD service

14:17:16.679 DEBUG BillCalculatorImpl#calcShares:24 - tip=$18.00, for $100.00 and
GOOD service
14:17:16.679 DEBUG BillCalculatorImpl#calcShares:33 - share=$29.50 for $100.00, 4
people and GOOD service

```

117.10. Alternative Test Slices

The `@SpringBootTest` annotation is a general purpose test annotation that likely will work in many generic cases. However, there are other cases where we may need a specific database or other technologies available. [Spring Boot pre-defines a set of Test Slices](#) that can establish more specialized test environments. The following are a few examples:

- `@DataJpaTest` - JPA/RDBMS testing for the data tier
- `@DataMongoTest` - MongoDB testing for the data tier
- `@JsonTest` - JSON data validation for marshalled data
- `@RestClientTest` - executing tests that perform actual HTTP calls for the web tier

We will revisit these topics as we move through the course and construct tests relative additional domains and technologies.

Chapter 118. Mocking Spring Boot Unit Integration Tests

In the previous `@SpringBootTest` example I showed you how to instantiate a complete Spring context to inject and execute test(s) against an integrated set of real components. However, in some cases we may need the Spring context—but do not need or want the interfacing components. In this example I am going to mock out the `TipCalculator` to produce whatever the test requires.

Example @SpringBoot/Mockito Definition

```
import org.springframework.boot.test.mock.mockito.MockBean;  
  
import static org.assertj.core.api.BDDAssertions.and;  
import static org.mockito.BDDMockito.given;  
import static org.mockito.BDDMockito.then;  
import static org.mockito.Mockito.times;  
  
@SpringBootTest(classes={BillCalculatorImpl.class})//defines custom Spring context ①  
@ActiveProfiles("test")  
@DisplayNameGeneration(DisplayNameGenerator.ReplaceUnderscores.class)  
@DisplayName("Bill CalculatorImpl Mocked Integration")  
public class BillCalculatorMockedNTest {  
    @Autowired //subject under test ②  
    private BillCalculator billCalculator;  
  
    @MockBean //will satisfy Autowired injection point within BillCalculatorImpl ③  
    private TipCalculator tipCalculatorMock;
```

① defining a custom context that excludes `TipCalculator` component(s)

② injecting `BillCalculator` bean under test from Spring context

③ defining a mock to be injected into `BillCalculatorImpl` in Spring context

118.1. Example @SpringBoot/Mockito Test

The actual test is similar to the earlier example when we injected a real `TipCalculator` from the Spring context. However, since we have a mock in this case we must define its behavior and then optionally determine if it was called.

Example @SpringBoot/Mockito Test

```
@Test  
public void calc_shares_for_people_including_tip() {  
    //given - we have a bill for 4 people and tip calculator that returns tip amount  
    BigDecimal billTotal = BigDecimal.valueOf(100.0);  
    ServiceQuality service = ServiceQuality.GOOD;  
    BigDecimal tip = billTotal.multiply(BigDecimal.valueOf(0.18));  
    int numPeople = 4;
```

```

//configure mock
given(tipCalculatorMock.calcTip(billTotal, service)).willReturn(tip); ①

//when - call method under test ②
BigDecimal shareResult = billCalculator.calcShares(billTotal, service, numPeople);

//then - tip calculator should be called once to get result
then(tipCalculatorMock).should(times(1)).calcTip(billTotal, service); ③

//verify correct result
BigDecimal expectedShare = billTotal.add(tip).divide(BigDecimal.valueOf(numPeople));
and.then(shareResult).isEqualTo(expectedShare); ④
}

```

① instruct the Mockito mock to return a tip result

② call method on subject under test

③ verify mock was invoked N times with the value of the bill and service

④ verify with AssertJ that the resulting share value was the expected share value

Chapter 119. Maven Unit Testing Basics

At this point we have some technical basics for how tests are syntactically expressed. Now lets take a look at how they fit into a module and how we can execute them as part of the Maven build.

You learned in earlier lessons that production artifacts that are part of our deployed artifact are placed in `src/main` (`java` and `resources`). Our test artifacts are placed in `src/test` (`java` and `resources`). The following example shows the layout of the module we are currently working with.

Example Module Test Source Tree

```
|-- pom.xml
`-- src
  '-- test
    |-- java
    |  '-- info
    |    '-- ejava
    |      '-- examples
    |        '-- app
    |          '-- testing
    |            '-- testbasics
    |              |-- PeopleFactory.java
    |              |-- jupiter
    |                |-- AspectJAssertionsTest.java
    |                |-- AssertionsTest.java
    |                |-- ExampleJUnit5Test.java
    |                '-- HamcrestAssertionsTest.java
    |              |-- mockito
    |                '-- ExampleMockitoTest.java
    |              |-- tips
    |                |-- BillCalculatorContractTest.java
    |                |-- BillCalculatorImplTest.java
    |                |-- BillCalculatorMockedNTest.java
    |                |-- BillCalculatorNTest.java
    |                '-- StandardTippingCalculatorImplTest.java
    |              '-- vintage
    |                '-- ExampleJUnit4Test.java
    '-- resources
    |-- application-test.properties
```

119.1. Maven Surefire Plugin

The [Maven Surefire plugin](#) looks for classes that have been compiled from the `src/test/java` source tree that have a [prefix of "Test"](#) or [suffix of "Test", "Tests", or "TestCase"](#) by default. Surefire starts up the JUnit context(s) and provides test results to the console and target/surefire-reports directory.

Surefire is part of the standard "jar" profile we use for normal Java projects and will run automatically. The following shows the final output after running all the unit tests for the module.

Example Surefire Execution of All Example Unit Tests

```
$ mvn clean test
...
[INFO] Results:
[INFO]
[INFO] Tests run: 24, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 14.280 s
```

Consult online documentation on how Maven Surefire can be configured. However, I will demonstrate at least one feature that allows us to filter tests executed.

119.2. Filtering Tests

One new JUnit Jupiter feature is the ability to categorize tests using `@Tag` annotations. The following example shows a unit integration test annotated with two tags: "springboot" and "tips". The "springboot" tag was added to all tests that launch the Spring context. The "tips" tag was added to all tests that are part of the tips example set of components.

Example @Tag

```
import org.junit.jupiter.api.*;
...
@SpringBootTest(classes = {BillCalculatorImpl.class}) //defining custom Spring context
@Tag("springboot") @Tag("tips") ①
...
public class BillCalculatorMockedNTest {
```

① test case has been tagged with JUnit "springboot" and "tips" tag values

119.3. Filtering Tests Executed

We can use the tag names as a "groups" property specification to Maven Surefire to only run matching tests. The following example requests all tests tagged with "tips" but not tagged with "springboot" are to be run. Notice we have fewer tests executed and a much faster completion time.

Filtering Tests Executed using Surefire Groups

```
$ mvn clean test -Dgroups='tips & !springboot' -Pbdd ① ②
...
[INFO] -----
[INFO] T E S T S
[INFO] -----
[INFO] Running Bill Calculator Contract
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.41 s - in
```

```

Bill Calculator Contract
[INFO] Running Bill CalculatorImpl
15:43:47.605 [main] DEBUG
info.ejava.examples.app.testing.testbasics.tips.BillCalculatorImpl - tip=$50.00, for
$100.00 and GOOD service
15:43:47.608 [main] DEBUG
info.ejava.examples.app.testing.testbasics.tips.BillCalculatorImpl - share=$37.50 for
$100.00, 4 people and GOOD service
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.165 s - in
Bill CalculatorImpl
[INFO] Running Standard Tipping Calculator
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.004 s - in
Standard Tipping Calculator
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 4.537 s

```

- ① execute tests with tag "tips" and without tag "springboot"
- ② activating "bdd" profile that configures Surefire reports within the example Maven environment setup to understand display names

119.4. Maven Failsafe Plugin

The [Maven Failsafe plugin](#) looks for classes compiled from the `src/test/java` tree that have a [prefix of "IT" or suffix of "IT", or "ITCase"](#) by default. Like Surefire, Failsafe is part of the standard Maven "jar" profile and runs later in the build process. However, unlike Surefire that runs within one [Maven phase \(test\)](#), Failsafe runs within the scope of four Maven phases: [pre-integration-test](#), [integration-test](#), [post-integration-test](#), and [verify](#)

- **pre-integration-test** - when external resources get started (e.g., web server)
- **integration-test** - when tests are executed
- **post-integration-test** - when external resources are stopped/cleaned up (e.g., shutdown web server)
- **verify** - when results of tests are evaluated and build potentially fails

119.5. Failsafe Overhead

Aside from the integration tests, all other processes are normally started and stopped through the use of Maven plugins. Multiple phases are required for IT tests so that:

- all resources are ready to test once the tests begin

- all resources can be shutdown prior to failing the build for a failed test

With the robust capability to stand up a Spring context within a single JVM, we really have limited use for Failsafe for testing Spring Boot applications. The exception for that is when we truly need to interface with something external—like stand up a real database or host endpoints in Docker images. I will wait until we get to topics like that before showing examples. Just know that when Maven references "integration tests", they come with extra hooks and overhead that may not be technically needed for integration tests—like the ones we have demonstrated in this lesson—that can be executed within a single JVM.

Chapter 120. @TestConfiguration

Tests often require additional components that are not part of the Spring context under test—or need to override one or more of those components. SpringBoot supplies a `@TestConfiguration` annotation that:

- allows the class to be skipped by standard component scan
- is loaded into a `@SpringBootTest` to add or replace components

120.1. Example Spring Context

In our example Spring context, we will have a `TipCalculator` component located using a component scan. It will have the name "standardTippingImpl" if we do not supply an override in the `@Component` annotation.

Example standardTippingImpl Bean

```
@Primary ①
@Component
public class StandardTippingImpl implements TipCalculator {
```

① declaring type as primary to make example more significant

That bean gets injected into `BillCalculatorImpl.tipCalculator` because it implements the required type.

Example Injection Target for Bean

```
@Component
@RequiredArgsConstructor
public class BillCalculatorImpl implements BillCalculator {
    private final TipCalculator tipCalculator;
```

120.2. Test TippingCalculator

Our intent here is to manually write a stub and have it replace the `TipCalculator` from the application's Spring context.

```
import org.springframework.boot.test.context.TestConfiguration;
...
@TestConfiguration(proxyBeanMethods = false) //skipped in component scan -- manually
included ①
public class MyTestConfiguration {
    @Bean
    public TipCalculator standardTippingImpl() { ②
        return new TipCalculator() {
```

```

    @Override
    public BigDecimal calcTip(BigDecimal amount, ServiceQuality
serviceQuality) {
        return BigDecimal.ZERO; ③
    }
};

}
}

```

① `@TestConfiguration` annotation prevents class from being picked up in normal component scan

② `standardTippingImpl` name matches existing component

③ test-specific custom response

120.3. Enable Component Replacement

Since we are going to replace an existing component, we need to enable bean overrides using the following property definition.

Enable Bean Override

```

@SpringBootTest(
    properties = "spring.main.allow-bean-definition-overriding=true"
)
public class TestConfigurationNTest {

```

Otherwise, we end up with the following error when we make our follow-on changes.

Bean Override Error Message

```

*****
APPLICATION FAILED TO START
*****

```

Description:

The bean 'standardTippingImpl', defined in class path resource
`[.../testconfiguration/MyTestConfiguration.class]`, could not be registered.
A bean with that name has already been defined in file
`[.../tips/StandardTippingImpl.class]` and overriding is disabled.

Action:

Consider renaming one of the beans or enabling overriding by setting
`spring.main.allow-bean-definition-overriding=true`

120.4. Embedded TestConfiguration

We can have the `@TestConfiguration` class automatically found using an embedded **static** class.

Embedded TestConfiguration

```
@SpringBootTest(properties={"..."})
public class TestConfigurationNTest {
    @Autowired
    BillCalculator billCalculator; ①

    @TestConfiguration(proxyBeanMethods = false)
    static class MyEmbeddedTestConfiguration { ②
        @Bean
        public TipCalculator standardTippingImpl() { ... }
    }
}
```

① injected `billCalculator` will be injected with `@Bean` from `@TestConfiguration`

② embedded static class used automatically

120.5. External TestConfiguration

Alternatively, we can place the configuration in a separate/stand-alone class.

```
@TestConfiguration(proxyBeanMethods = false)
public class MyTestConfiguration {
    @Bean
    public TipCalculator tipCalculator() {
        return new TipCalculator() {
            @Override
            public BigDecimal calcTip(BigDecimal amount, ServiceQuality
serviceQuality) {
                return BigDecimal.ZERO;
            }
        };
    }
}
```

120.6. Using External Configuration

The external `@TestConfiguration` will only be used if specifically named in either:

- `@SpringBootTest.classes`
- `@ContextConfiguration.classes`
- `@Import.value`

Pick one way.

Imported TestConfiguration

```
@SpringBootTest(  
    classes=MyTestConfiguration.class, //way1 ①  
    properties = "spring.main.allow-bean-definition-overriding=true"  
)  
@ContextConfiguration(classes=MyTestConfiguration.class) //way2 ②  
@Import(MyTestConfiguration.class) //way3 ③  
public class TestConfigurationNTest {
```

① way1 leverages the `@SpringBootTest configuration

② way2 pre-dates `@SpringBootTest`

③ way3 pre-dates `@SpringBootTest` and is a standard way to import a configuration definition from one class to another

120.7. TestConfiguration Result

Running the following test results in:

- a single `TipCalculator` registered in the list because each considered have the same name and overriding is enabled
- the `TipCalculator` used is one of the `@TestConfiguration`-supplied components

TipCalculator Replaced by @TestConfiguration-supplied Component

```
@SpringBootTest(  
    classes=MyTestConfiguration.class,  
    properties = "spring.main.allow-bean-definition-overriding=true")  
public class TestConfigurationNTest {  
    @Autowired  
    BillCalculator billCalculator;  
    @Autowired  
    List<TipCalculator> tipCalculators;  
  
    @Test  
    void calc_has_been_replaced() {  
        //then  
        then(tipCalculators).as("too many topCalculators").hasSize(1);  
        then(tipCalculators.get(0).getClass()).hasAnnotation(TestConfiguration.class);  
    } ①  
}
```

① `@Primary TipCalculator` bean replaced by our `@TestConfiguration`-supplied bean

Chapter 121. Summary

In this module we:

- learned the importance of testing
- introduced some of the testing capabilities of libraries integrated into `spring-boot-starter-test`
- went thru an overview of JUnit Vintage and Jupiter test constructs
- stressed the significance of using assertions in testing and the value in making them based on natural-language to make them easy to understand
- introduced how to inject a mock into a subject under test
- demonstrated how to define a mock for testing a particular scenario
- demonstrated how to inspect calls made to the mock during testing of a subject
- discovered how to switch default Mockito and AssertJ methods to match Business-Driven Development (BDD) acceptance test keywords
- implemented unit integration tests with Spring context using `@SpringBootTest`
- implemented mocks into the Spring context of a unit integration test
- ran tests using Maven Surefire
- implemented a `@TestConfiguration` with component override

AutoRentals Assignment 1

copyright © 2024 jim stafford (jim.stafford@jhu.edu)

- 2024/09/20 - corrected ToolRentalProperties/BoatRentalProperties typo on ConfigurationProperties assignment

The following three areas (Config, Logging, and Testing) map out the different portions of "Assignment 1". It is broken up to provide some focus.

- Each of the areas (1a Config, 1b Logging, and 1c Testing) are separate but are to be turned in together, under a single root project tree. There is no relationship between the classes used in the three areas—even if they have the same name. Treat them as separate.
- Each of the areas are further broken down into parts. The parts of the Config area are separate. Treat them that way by working in separate module trees (under a common grandparent). The individual parts for Logging and Testing overlap. Once you have a set of classes in place—you build from that point. They should be worked/turned in as a single module each (one for Logging and one for Testing; under the same parent as Config).

A set of starter projects is available in [assignment-starter/autorentals-starter](#). It is expected that you can implement the complete assignment on your own. However, the Maven poms and the portions unrelated to the assignment focus are commonly provided for reference to keep the focus on each assignment part. Your submission should not be a direct edit/hand-in of the starters. Your submission should—at a minimum:

- use your own Maven groupIds
- use your own Java package names below a given base
- extend either [spring-boot-starter-parent](#) or [ejava-build-parent](#)

Your assignment submission should be a single-rooted source tree with sub-modules or sub-module trees for each independent area part. The assignment starters—again can be your guide for mapping these out.

Example Project Layout

```
|-- assignment1-autorentals-autoconfig
|   |-- pom.xml
|   |-- rentals-autoconfig-app
|   |-- rentals-autoconfig-toolrentals
|       '-- rentals-autoconfig-starter
|-- assignment1-autorentals-beanfactory # 1st
|   |-- pom.xml
|   |-- rentals-beanfactory-app
|   |-- rentals-beanfactory-autorentals
|       '-- rentals-beanfactory-iface
|-- assignment1-autorentals-configprops
```

```
|   |-- pom.xml
|   '-- src
|-- assignment1-autorentals-logging
|   |-- pom.xml
|   '-- src
|-- assignment1-autorentals-propertysource
|   |-- pom.xml
|   '-- src
|-- assignment1-autorentals-testing
|   |-- pom.xml
|   '-- src
`-- pom.xml  <== your project root (separate from course examples tree)
```

Chapter 122. Assignment 1a: App Config

122.1. @Bean Factory Configuration

122.1.1. Purpose

In this portion of the assignment, you will demonstrate your knowledge of configuring a decoupled application integrated using Spring Boot. You will:

1. implement a service interface and implementation component
2. package a service within a Maven module separate from the application module
3. implement a Maven module dependency to make the component class available to the application module
4. use a `@Bean` factory method of a `@Configuration` class to instantiate a Spring-managed component

122.1.2. Overview

In this portion of the assignment you will be implementing a component class and defining that as a Spring bean using a `@Bean` factory located within the core application JAR.

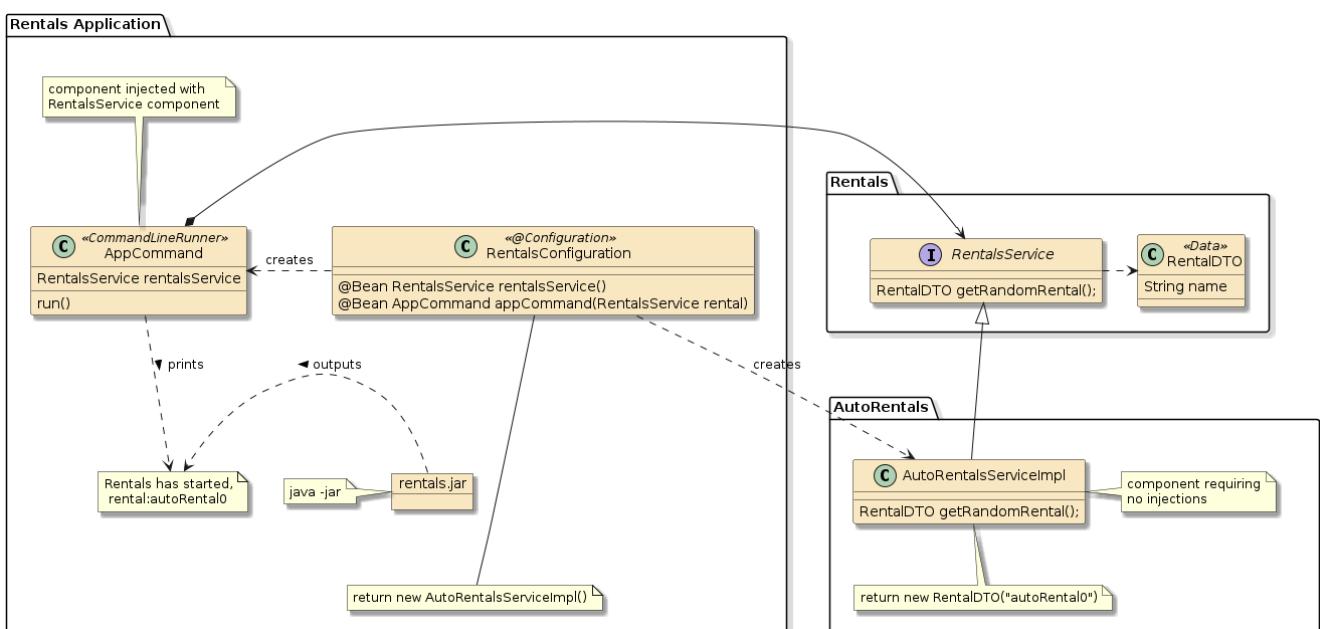


Figure 31. `@Bean` Factory Configuration

122.1.3. Requirements

1. Create an interface module with
 - a `RentalDTO` class with `name` property. This is a simple data class.
 - a `RentalsService` interface with a `getRandomRental()` method. This method returns a single `RentalDTO` instance.
2. Create an AutoRental implementation module with

- a. AutoRental implementation implementing the `RentalsService` interface that returns a `RentalDTO` with a name (e.g., "autoRental0").
3. Create an application module with
- a. a class that
 - i. implements `CommandLineRunner` interface
 - ii. has the `RentalsService` component injected using **constructor injection**
 - iii. a `run()` method that
 - A. calls the `RentalsService` for a random AutoRentalDTO
 - B. prints a startup message with the DTO name
 - C. relies on a `@Bean` factory to register it with the container and **not** a `@Component` mechanism
 - b. a `@Configuration` class with two `@Bean` factory methods
 - i. one `@Bean` factory method to instantiate a `RentalsService` autoRental implementation
 - ii. one `@Bean` factory method to instantiate the `AppCommand` injected with a `RentalsService` bean (not a POJO)

`@Bean` factories that require external beans, can have the dependencies injected by declaring them in their method signature. Example:



```
TypeA factoryA() {return new TypeA(); }
TypeB factoryB(TypeA beanA) {return new TypeB(beanA); }
```

That way the you can be assured that the dependency is a fully initialized bean versus a partially initialized POJO.

- c. a `@SpringBootApplication` class that initializes the Spring Context—which will process the `@Configuration` class
4. Turn in a source tree with three or more complete Maven modules that will build and demonstrate a configured Spring Boot application.

122.1.4. Grading

Your solution will be evaluated on:

1. implement a service interface and implementation component
 - a. whether an interface module was created to contain interface and data dependencies of that interface
 - b. whether an implementation module was created to contain a class implementation of the interface
2. package a service within a Maven module separate from the application module
 - a. whether an application module was created to house a `@SpringBootApplication` and

`@Configuration` set of classes

3. implement a Maven module dependency to make the component class available to the application module
 - a. whether at least three separate Maven modules were created with a one-way dependency between them
4. use a `@Bean` factory method of a `@Configuration` class to instantiate Spring-managed components
 - a. whether the `@Configuration` class successfully instantiates the `RentalsService` component
 - b. whether the `@Configuration` class successfully instantiates the `AppCommand` component injected with a `RentalsService` component.

122.1.5. Additional Details

1. The `spring-boot-maven-plugin` can be used to both build the Spring Boot executable JAR and execute the JAR to demonstrate the instantiations, injections, and desired application output.
2. A quick start project is available in [`assignment-starter/autorentals-starter/assignment1-autorentals-beanfactory`](#). Modify Maven groupId and Java package if used.

122.2. Property Source Configuration

122.2.1. Purpose

In this portion of the assignment, you will demonstrate your knowledge of how to flexibly supply application properties based on application, location, and profile options. You will:

1. implement value injection into a Spring Component
2. define a default value for the injection
3. specify property files from different locations
4. specify a property file for a basename
5. specify properties based on an active profile
6. specify a both straight properties and YAML property file sources

122.2.2. Overview

You are given a Java application that prints out information based on injected properties, defaults, a base property file, and executed using different named profiles. You are to supply several profile-specific property files that — when processed together — produce the required output.

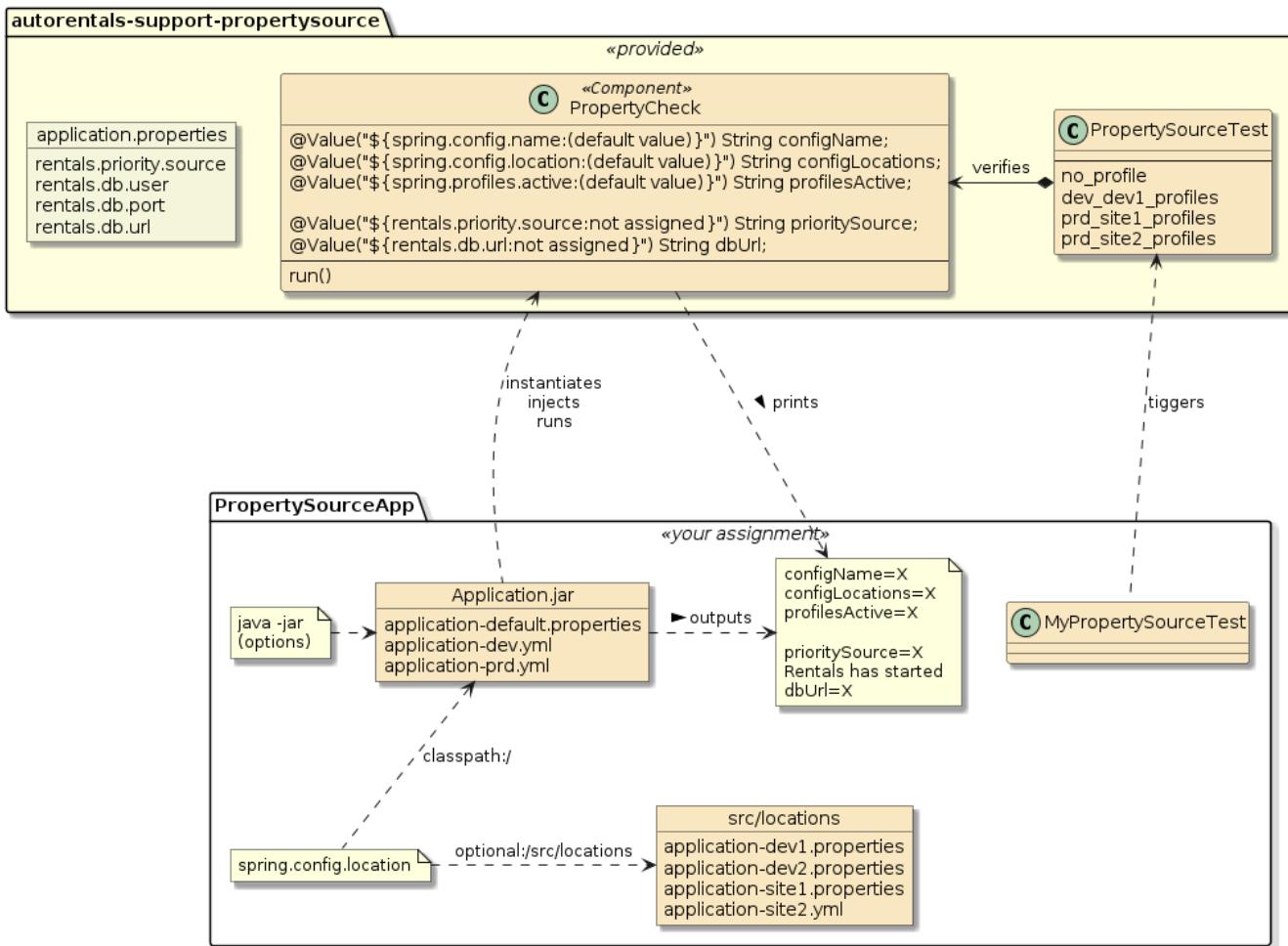


Figure 32. Property Source Configuration

This assignment involves no new Java coding (the "assignment starter" has all you need). It is designed as a puzzle where—given some constant surroundings—you need to determine what properties to supply and in which file to supply them, to satisfy all listed test scenarios.

The assignment is structured into two modules: app and support

- app - is your assignment. The skeletal structure is provided in [autorentals-starter/assignment1-autorentals-propertysource](#)
- support - is provided is provided in the [autorentals-starter/autorentals-support-propertysource](#) module and is to be used, unmodified through a Maven dependency. It contains a default `application.properties` file with skeletal values, a component that gets injected with property values, and a unit integration test that verifies the program results.

The `autorentals-support-propertysource` module provides the following resources.

PropertyCheck Class

This class has property injections defined with default values when they are not supplied. This class will be in your classpath and automatically packaged within your JAR.

Supplied Component Class

```
public class PropertyCheck implements CommandLineRunner {
    @Value("${spring.config.name:(default value)}") String configName;
```

```

@Value("${spring.config.location:(default value)}") String configLocations;
@Value("${spring.profiles.active:(default value)}") String profilesActive;

@Value("${rentals.priority.source:not assigned}") String prioritySource;
@Value("${rentals.db.url:not assigned}") String dbUrl;

```

application.properties File

This file provides a template of a database URL with placeholders that will get populated from other property sources. This file will be in your classpath and automatically packaged within your JAR.

Supplied application.properties File

```

#application.properties
rentals.priority.source=application.properties
rentals.db.user=user
rentals.db.port=0000
①
rentals.db.url=mongodb://${rentals.db.user}:${rentals.db.password}@${rentals.db.host}:${rentals.db.port}/test?authSource=admin

```

① `rentals.db.url` is built from several property placeholders. `password` is not specified.

PropertySourceTest Class

a unit integration test is provided that can verify the results of your property file population. This test will run automatically during the Maven build.

Supplied Test Class

```

public class PropertySourceTest {
    static final String CONFIG_LOCATION="classpath:/,optional:file:src/locations/";
    class no_profile {
        @Test
        void has_expected_sources() throws Exception {
        @Test
        void has_generic_files_in_classpath() {
        @Test
        void has_no_credential_files_in_classpath() {
        @Test
        void sources_have_unique_properties() {
    class dev_dev1_profiles {
        @Test
        void has_expected_sources() throws Exception {
        @Test
        void sources_have_unique_properties() {
    class dev_dev2_profiles {
        @Test
        void has_expected_sources() throws Exception {
        @Test
        void sources_have_unique_properties() {

```

```

class prd_site1_profiles {
    @Test
    void has_expected_sources() throws Exception {
        @Test
        void sources_have_unique_properties() {
class prd_site2_profiles {
    @Test
    void has_expected_sources() throws Exception {
        @Test
        void sources_have_unique_properties() {
class dev_dev1_dev2 {
    @Test
    void sources_have_unique_properties() {
class prd_site1_site2 {
    @Test
    void sources_have_unique_properties() {

```

122.2.3. Requirements



The starter module has much of the setup already defined.

1. Create a dependency on the support module. (provided in starter)

Your Maven Dependency on Provided Assignment Artifacts

```

<dependency>
    <groupId>info.ejava.assignments.propertysource.autorentals</groupId>
    <artifactId>autorentals-support-propertysource</artifactId>
    <version>${ejava.version}</version>
</dependency>

```

2. Add a `@SpringBootApplication` class with `main()` (provided in starter)

Your `@SpringBootApplication` Class

```

package info.ejava_student.starter.assignment1.propertysource.rentals;

import info.ejava.assignments.propertysource.rentals.PropertyCheck;

@SpringBootApplication
public class PropertySourceApp {

```

3. Provide the following property file sources. (provided in starter) `application.properties` will be provided through the dependency on the support module and will get included in the JAR.

Your Property Classpath and File Path Files

```

src/main/resources:/ ①
    application-default.properties

```

```
application-dev.yml ③  
application-prd.properties  
src/locations/ ②  
  application-dev1.properties  
  application-dev2.properties  
  application-site1.properties  
  application-site2.yml ③
```

① `src/main/resources` files will get packaged into JAR and will automatically be in the classpath at runtime

② `src/locations` are not packaged into the JAR and will be referenced by a command-line parameter to add them to the config location path

③ example uses of YAML files



yml files must be expressed as a YAML file

application-dev.yml and application-site2.yml must be expressed using YAML syntax

4. Enable the unit integration test from the starter when you are ready to test—by removing `@Disabled`.

```
package info.ejava_student.starter.assignment1.propertysource.rentals;  
  
import info.ejava.assignments.propertysource.rentals.PropertySourceTest;  
...  
//we will cover testing in a future topic, very soon  
@Disabled //enable when ready to start assignment  
public class MyPropertySourceTest extends PropertySourceTest {
```

5. Use a constant base command. This part of the command remains constant.

```
$ mvn clean package -DskipTests=true ②  
$ java -jar target/*-propertysource-1.0-SNAPSHOT-bootexec.jar  
--spring.config.location=classpath:/,optional:file:src/locations/ ①
```

① this is the base command for 5 specific commands that specify profiles active

② need to skip tests to build JAR before complete

The only modification to the command line will be the conditional addition of a profile activation.

```
--spring.profiles.active= ①
```

① the following 5 commands will supply a different value for this property

6. Populate the property and YAML files so that the scenarios in the following paragraph are

satisfied. The default starter with the "base command" and "no active profile" set, produces the following by default.

```
$ java -jar target/*-propertysource-1.0-SNAPSHOT-bootexec.jar  
--spring.config.location=classpath:/,optional:file:src/locations/  
  
configName=(default value)  
configLocations=classpath:/,optional:file:src/locations/  
profilesActive=(default value)  
prioritySource=application-default.properties  
Rentals has started  
dbUrl=mongodb://user:NOT_SUPPLIED@NOT_SUPPLIED:0000/test?authSource=admin
```



Any property value that does not contain a developer/site-specific value (e.g., defaultUser and defaultPass) must be provided by a property file packaged into the JAR (i.e., source **src/main/resources**)



Any property value that does contain a developer/site-specific value (e.g., dev1pass and site1Pass) must be provided by a property file in the **file:** part of the location path and not in the JAR (i.e., source **src/locations**).

Complete the following 5 scenarios:

a. No Active Profile Command Result

```
configName=(default value)  
configLocations=classpath:/,optional:file:src/locations/  
profilesActive=(default value)  
prioritySource=application-default.properties  
Rentals has started  
dbUrl=mongodb://defaultUser:defaultPass@defaulthost:27027/test?authSource=admin
```



You must supply a populated set of configuration files so that, under this option, **user:NOT_SUPPLIED@NOT_SUPPLIED:0000** becomes **defaultUser:defaultPass@defaulthost:27027**.

b. dev,dev1 Active Profile Command Result

```
--spring.profiles.active=dev,dev1
```

```
configName=(default value)  
configLocations=classpath:/,optional:file:src/locations/  
profilesActive=dev,dev1  
prioritySource=application-dev1.properties  
Rentals has started
```

```
dbUrl=mongodb://devUser:dev1pass@127.0.0.1:11027/test?authSource=admin
```

c. dev,dev2 Active Profile Command Result

```
--spring.profiles.active=dev,dev2
```

```
configName=(default value)
configLocations=classpath:/,optional:file:src/locations/
profilesActive=dev,dev2
prioritySource=application-dev2.properties
Rentals has started
dbUrl=mongodb://devUser:dev2pass@127.0.0.1:22027/test?authSource=admin
```



The development profiles share the same user and host.

d. prd,site1 Active Profile Command Result

```
--spring.profiles.active=prd,site1
```

```
configName=(default value)
configLocations=classpath:/,optional:file:src/locations/
profilesActive=prd,site1
prioritySource=application-site1.properties
Rentals has started
dbUrl=mongodb://prdUser:site1pass@db.site1.net:27017/test?authSource=admin
```

e. prd,site2 Active Profile Command Result

```
--spring.profiles.active=prd,site2
```

```
configName=(default value)
configLocations=classpath:/,optional:file:src/locations/
profilesActive=prd,site2
prioritySource=application-site2.properties
Rentals has started
dbUrl=mongodb://prdUser:site2pass@db.site2.net:27017/test?authSource=admin
```



The production/site profiles share the same user and port.

7. No property with the same value may be present in multiple property files. You must make use of property source inheritance when requiring a common value.

8. Turn in a source tree with a complete Maven module that will build and demonstrate the `@Value` injections for the different active profile settings.

122.2.4. Grading

Your solution will be evaluated on:

1. implement value injection into a Spring Component
 - a. whether `@Component` attributes were injected with values from property sources
2. define a default value for the injection
 - a. whether default values were correctly accepted or overridden
 - b. whether each unique property value was expressed in a single source file and property file inheritance was used when common values were needed
3. specify property files from different locations
 - a. whether your solution provides property values coming from multiple file locations
 - i. any property value that does not contain a developer/site-specific value (e.g., `defaultUser` and `defaultPass`) must be provided by a property file within the JAR
 - ii. any property value that contains developer/site-specific values (e.g., `dev1pass` and `site1pass`) must be provided by a property file outside of the JAR
 - b. the given `application.properties` file may not be modified
 - c. named `.properties` files are supplied as properties files
 - d. named `.yml` (i.e., `application-dev.yml`) files are supplied as YAML files
4. specify properties based on an active profile
 - a. whether your output reflects current values for `dev1`, `dev2`, `site1`, and `site2` profiles
5. specify both straight properties and YAML property file sources
 - a. whether your solution correctly supplies values for at least 1 properties file
 - b. whether your solution correctly supplies values for at least 1 YAML file

122.2.5. Additional Details

1. The `spring-boot-maven-plugin` can be used to both build the Spring Boot executable JAR and demonstrate the instantiations, injections, and desired application output.
2. A quick start project is available in `assignment-starter/autorentals-starter/assignment1-autorentals-propertysource` that supplies much of the boilerplate file and Maven setup. Modify Maven groupId and Java package if used.
3. An integration unit test (`PropertySourceTest`) is provided within the support module that can automate the verifications.
4. Ungraded Question to Ponder: How could you at runtime, provide a parameter option to the application to make the following output appear?

Alternate Output

```
configName=autorentals
configLocations=(default value)
profilesActive=(default value)
prioritySource=not assigned
Rentals has started
dbUrl=not assigned
```

122.3. Configuration Properties

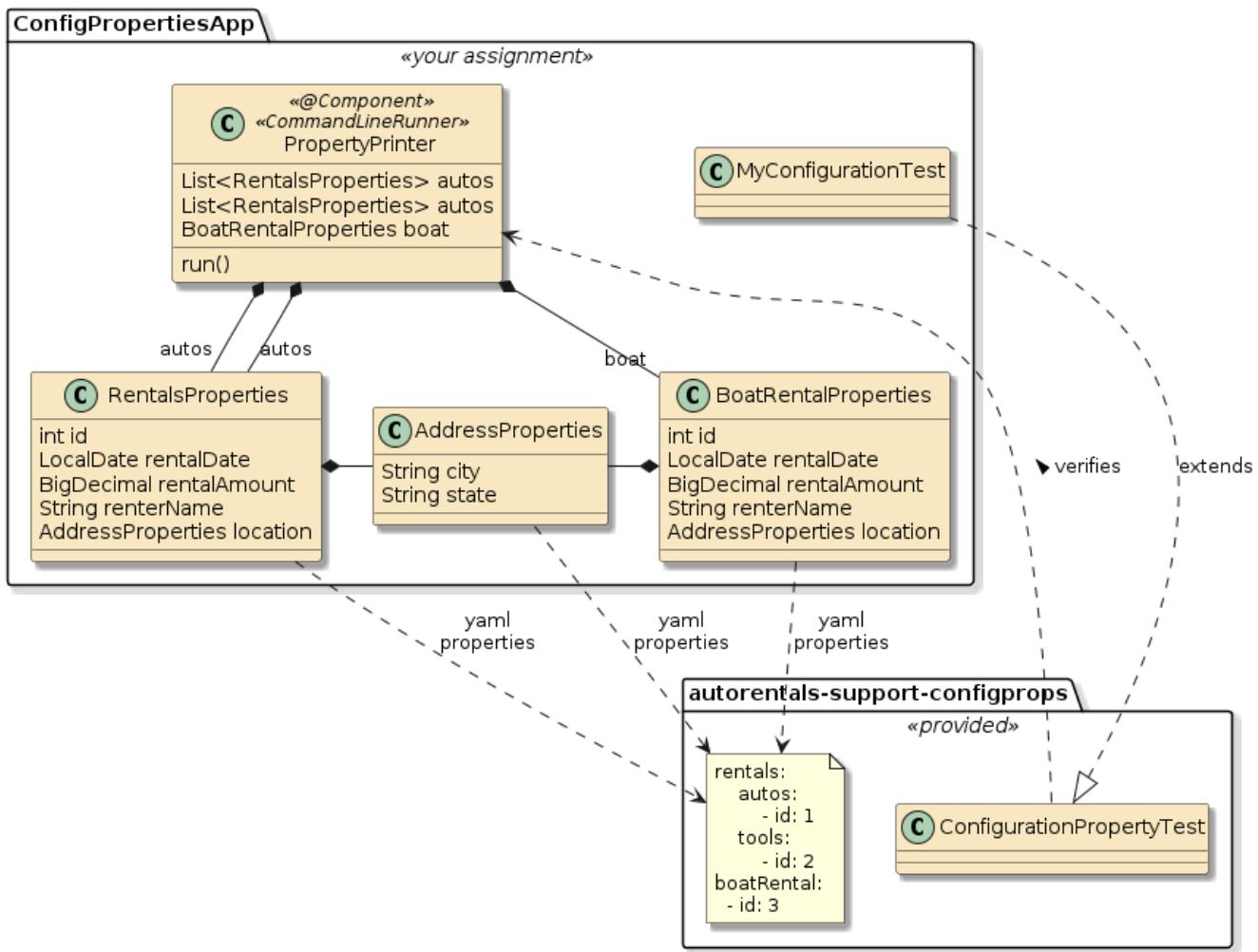
122.3.1. Purpose

In this portion of the assignment, you will demonstrate your knowledge of injecting properties into a `@ConfigurationProperties` class to be injected into components - to encapsulate the runtime configuration of the component(s). You will:

1. map a Java `@ConfigurationProperties` class to a group of properties
2. create a read-only `@ConfigurationProperties` class using `@ConstructorBinding`
3. define a Jakarta EE Java validation rule for a property and have the property validated at runtime
4. generate boilerplate JavaBean methods using Lombok library
5. map nested properties to a `@ConfigurationProperties` class
6. reuse a `@ConfigurationProperties` class to map multiple property trees of the same structure
7. use `@Qualifier` annotation and other techniques to map or disambiguate an injection

122.3.2. Overview

In this assignment, you are going to finish mapping a YAML file of properties to a set of Java classes and have them injected as `@ConfigurationProperty` beans.



`BoatRentalProperties` is a straight-forward, single use bean that can have the class directly mapped to a specific property prefix. `RentalsProperties` will be mapped to two separate prefixes—so the mapping cannot be applied directly to that class. Keep this in mind when wiring up your solution.

An integration unit test is supplied and can be activated when you are ready to test your progress.

122.3.3. Requirements

- Given the following read-only property classes, application.yml file, and `@Component` ...
 - read-only property classes

```

@Value
public class RentalProperties {
    private int id;
    private LocalDate rentalDate;
    private BigDecimal rentalAmount;
    private String renterName;
    private AddressProperties location;
}
  
```

```

@Value
public class BoatRentalProperties { }
  
```

```
    private int id;
    private LocalDate rentalDate;
    private BigDecimal rentalAmount;
    private String renterName;
    private AddressProperties location;
}
```

```
@Value
public class AddressProperties {
    private final String city;
    private final String state;
}
```



Lombok `@Value` annotation defines the class to be read-only by only declaring getter()s and no setter()s. This will require use of constructor binding.

The property classes are supplied in the starter module.

b. `application.yml` YAML file

```
rentals:
  autos:
    - id: 1
      rentalDate: 2010-07-01 ⑥
      rentalAmount: 100.00 ①
      renterName: Joe Camper
      location:
        city: Jonestown
        state: PA
    #...
  tools:
    - id: 2
      rental-date: 2000-01-01
      rental-amount: 1000 ②
      renter_name: Itis Clunker ③
      location:
        city: Dundalk
        state: MD
boatRental:
  id: 3
  RENTAL_DATE: 2022-08-01 ④
  RENTAL_AMOUNT: 200_000
  RENTER-NAME: Alexus Blabidy ⑤
  LOCATION:
    city: Annapolis
    state: MD
```

- ① lower camelCase
- ② lower kabob-case
- ③ lower snake_case
- ④ upper SNAKE-CASE
- ⑤ upper KABOB-CASE
- ⑥ LocalDate parsing will need to be addressed

The full contents of the YAML file can be found in the [autorentals-support/autorentals-support-configprops](#) support project. YAML was used here because it is easier to express and read the nested properties.

Notice that multiple text cases (upper, lower, snake, kabob) are used to map the the same Java properties. This demonstrates one of the benefits in using `@ConfigurationProperties` over `@Value` injection — configuration files can be expressed in syntax that may be closer to the external domain.

Note that the `LocalDate` will require additional work to parse. That is provided to you in the starter project and described later in this assignment.

- c. `@Component` with constructor injection and getters to inspect what was injected

```
//@Component
@Getter
@RequiredArgsConstructor
public class PropertyPrinter implements CommandLineRunner {
    private final List<RentalProperties> autos;
    private final List<RentalProperties> tools;
    private final BoatRentalProperties boat;

    @Override
    public void run(String... args) throws Exception {
        System.out.println("autos:" + format(autos));
        System.out.println("tools:" + format(tools));
        System.out.println("boat:" + format(null==boat ? null : List.of(boat)));
    }

    private String format(List<?> rentals) {
        return null==rentals ? "(null)" :
            String.format("%s", rentals.stream()
                .map(r->"*" + r.toString())
                .collect(Collectors.joining(System.lineSeparator(), System
                    .lineSeparator(), "")));
    }
}
```

The source for the `PropertyPrinter` component is supplied in the starter module. Except for getting it registered as a component, there should be nothing needing change here.

2. When running the application, a `@ConfigurationProperties` beans will be created to represent the contents of the YAML file as two separate `List<RentalProperties>` objects and a `BoatRentalProperties` object. When properly configured, they will be injected into the `@Component`, and it will output the following.

```
autos:  
*RentalsProperties(id=1, rentalDate=2010-07-01, rentalAmount=100.0, renterName=Joe  
Camper, location=AddressProperties(city=Jonestown, state=PA))  
*RentalsProperties(id=4, rentalDate=2021-05-01, rentalAmount=500000,  
renterName=Jill Suburb, location=AddressProperties(city=, state=MD)) ①  
*RentalPropertiesProperties(id=5, rentalDate=2021-07-01, rentalAmount=1000000,  
renterName=M.R. Bigshot, location=AddressProperties(city=Rockville, state=MD))  
autos:  
*RentalProperties(id=2, rentalDate=2000-01-01, rentalAmount=1000, renterName=Itis  
Clunker, location=AddressProperties(city=Dundalk, state=MD))  
boat:  
*BoatRentalProperties(id=3, rentalDate=2022-08-01, rentalAmount=200000,  
renterName=Alexus Blabidy, location=AddressProperties(city=Annapolis, state=MD))
```

① one of the autoRentals addresses is missing a city



The "assignment starter" supplies most of the Java code needed for the `PropertyPrinter`.

3. Configure your solution so that the `BoatRentalProperties` bean is injected into the `PropertyPrinter` component along with the List of auto and tool `RentalProperties`. There is a skeletal configuration supplied in the application class. Most of your work will be within this class.

```
@SpringBootApplication  
public class ConfigPropertiesApp {  
    public static void main(String[] args)  
  
        public List<RentalsProperties> autos() {  
            return new ArrayList<>();  
        }  
        public List<RentalsProperties> tools() {  
            return new ArrayList<>();  
        }  
}
```

4. Turn in a source tree with a complete Maven module that will build and demonstrate the configuration property processing and output of this application.

122.3.4. Grading

Your solution will be evaluated on:

1. map a Java `@ConfigurationProperties` class to a group of properties
 - a. whether Java classes were used to map values from the given YAML file
2. create a read-only `@ConfigurationProperties` class using `@ConstructorBinding`
 - a. whether read-only Java classes, using `@ConstructorBinding` were used to map values from the given YAML file
3. generate boilerplate JavaBean methods using Lombok library
 - a. whether lombok annotations were used to generate boilerplate Java bean code
4. map nested properties to a `@ConfigurationProperties` class
 - a. whether nested Java classes were used to map nested properties from a given YAML file
5. reuse a `@ConfigurationProperties` class to map multiple property trees of the same structure
 - a. whether multiple property trees were instantiated using the same Java classes
6. use `@Qualifier` annotation and other techniques to map or disambiguate an injection
 - a. whether multiple `@ConfigurationProperty` beans of the same type could be injected into a `@Component` using a disambiguating technique.

122.3.5. Additional Details

1. A starter project is available in [autorentals-starter/assignment1-autorentals-configprops](#). Modify Maven groupId and Java package if used.
1. The `spring-boot-maven-plugin` can be used to both build the Spring Boot executable JAR and demonstrate the instantiations, injections, and desired application output.
2. The support project contains an integration unit test that verifies the `PropertyPrinter` component was defined and injected with the expected data. It is activated through a Java class in the starter module. Activate it when you are ready to test.

```
//we will cover testing in a future topic, very soon
@Disabled //remove to activate when ready to test
public class MyConfigurationTest extends ConfigurationPropertyTest {
}
```

3. Ungraded Question to Ponder: What change(s) could be made to the application to validate the properties and report the following error?

Alternate Output

```
Binding validation errors on rentals.autos[1].location
...
codes [rentals.autos[1].location.city,city]; arguments []; default message [city];
default message [must not be blank]
```

122.4. Auto-Configuration

122.4.1. Purpose

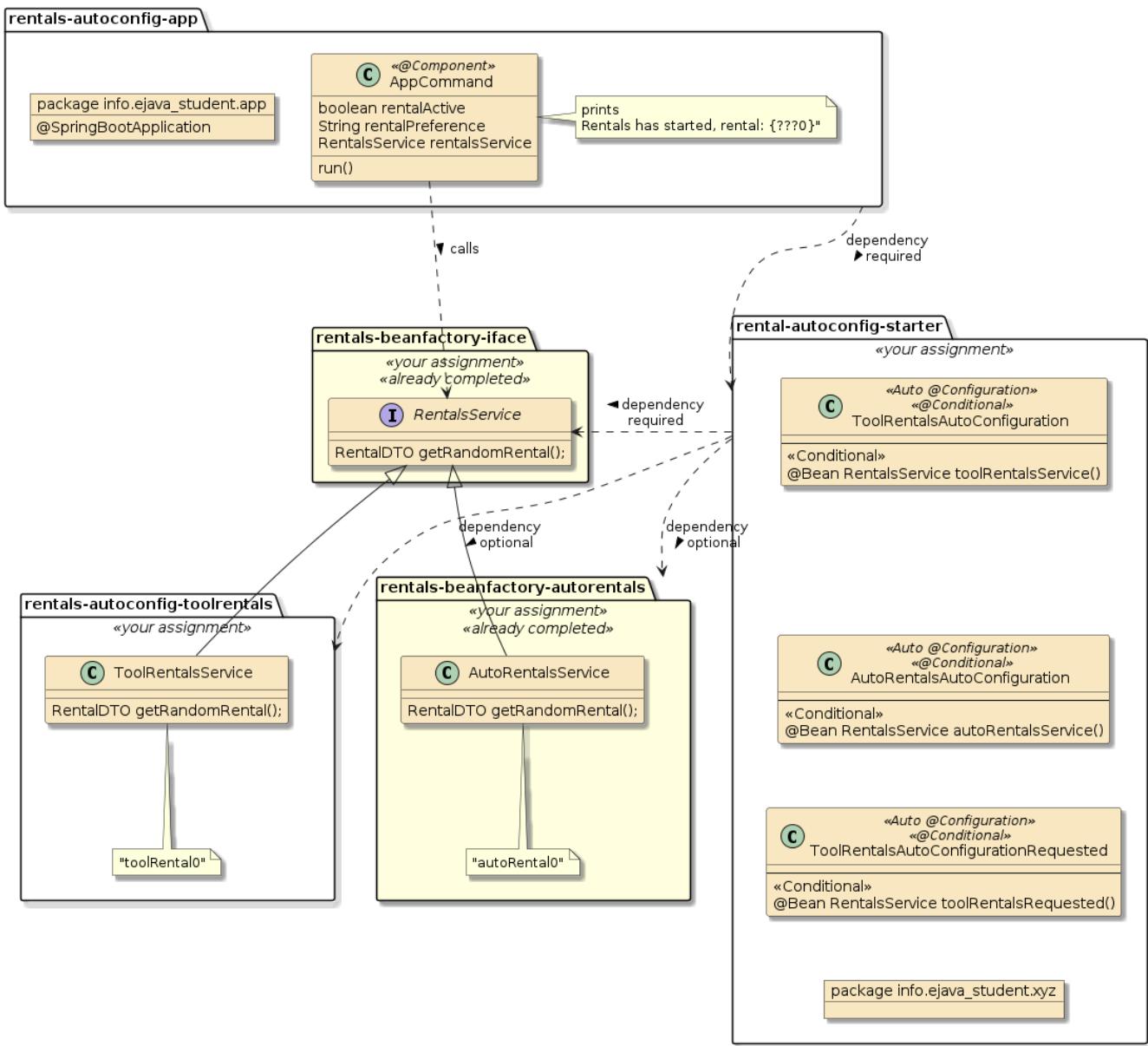
In this portion of the assignment, you will demonstrate your knowledge of developing `@Configuration` classes used for Auto-Configuration of an application.

You will:

1. Create a `@Configuration` class or `@Bean` factory method to be registered based on the result of a condition at startup
2. Create a Spring Boot Auto-configuration module to use as a "Starter"
3. Bootstrap Auto-configuration classes into applications using an `... AutoConfiguration.imports` metadata file
4. Create a conditional component based on the presence of a property value
5. Create a conditional component based on a missing component
6. Create a conditional component based on the presence of a class
7. Define a processing dependency order for Auto-configuration classes

122.4.2. Overview

In this assignment, you will be building a starter module, with a prioritized list of Auto-Configuration classes that will bootstrap an application depending on runtime environment. This application will have one (1) type of `RentalsService` out of a choice of two (2) based on the environment at runtime.



Make the `@SpringBootApplication` class package independent of `@Configuration` class packages



The Java package for the `@SpringBootApplication` class must **not** be a parent or at the same Java package as the `@Configuration` classes. Doing so, would place the `@Configuration` classes in the default component scan path and make them part of the core application — versus a conditional extension of the application.

122.4.3. Requirements



You have already implemented the `RentalsService` interface and `AutoRental` implementation modules in your Bean Factory solution. You will reuse them through a Maven dependency. `ToolRental` implementation is a copy of `AutoRental` implementation with name changes.

1. Create a `RentalsService` interface module (already completed for beanfactory)
 - a. Add an interface to return a random rental as a `RentalDTO` instance
2. Create a `AutoRental` implementation module (already completed for beanfactory)

- a. Add an implementation of the interface to return a `RentalDTO` with "auto" in the name property.
3. Create a ToolRentals implementation implementation module (new)
- a. Add an implementation of the interface to return a `RentalDTO` with "tool" in the name property.
4. Create an Application Module with a `@SpringBootApplication` class
- a. Add a `CommandLineRunner` implementation class that gets injected with a `RentalsService` bean and prints "Rentals has started" with the name of rental coming from the injected bean.
 - i. Account for a null implementation injected when there is no implementation such that it still quietly prints the state of the component.
 - ii. Include an injection (by any means) for properties `rentals.active` and `rentals.preference` to print their values. Account for when they are not supplied.
 - b. Add a `@Bean` factory for the `CommandLineRunner` implementation class—registered as "appCommand".
 - i. Make the injection of the `RentalsService` optional to account for when there is no implementation



`@Autowired(required=false)`

- c. Do not place any direct Java or Maven dependencies from the Application Module to the `RentalService` implementation modules.



At this point you are have mostly repeated the bean factory solution except that you have eliminated the `@Bean` factory for the `RentalsService` in the Application module, added a ToolRental implementation option, and removed a few Maven module dependencies.

5. Create a Rental starter Module

- a. Add a dependency on the `RentalsService` interface module
- b. Add a dependency on the `RentalsService` implementation modules and make them "**optional**" (this is **important**) so that the application module will need to make an explicit dependency on the implementation for them to be on the runtime classpath.
- c. Add three conditional Auto-configuration classes
 - i. one that provides a `@Bean` factory for the `ToolRentalsService` implementation class
 - A. Make this conditional on the presence of the `ToolRental` class(es) being available on the classpath
 - ii. one that provides a `@Bean` factory for the `AutoRentalsService` implementation class
 - A. Make this conditional on the presence of the `AutoRental` class(es) being available on the classpath
 - iii. A third that provides another `@Bean` factory for the `ToolRental` implementation class
 - A. Make this conditional on the presence of the `ToolRental` class(es) being available on

the classpath

- B. Make this also conditional on the property `rentals.preference` having the value of `tools`.
- d. Set the following priorities for the Auto-configuration classes
 - i. make the ToolRental/property (3rd from above) the highest priority
 - ii. make the AutoRental factory the next highest priority
 - iii. make the ToolRental factory the lowest priority



You can use `org.springframework.boot.autoconfigure.AutoConfigureOrder` to set a relative order — with the lower value having a higher priority.

- e. Disable all RentalsService implementation `@Bean` factories if the property `rentals.active` is present and has the value `false`



Treat `false` as being not the value `true`. Spring Boot does not offer a disable condition, so you will be looking to enable when the property is `true` or missing.

- f. Perform necessary registration steps within the Starter module to make the Auto-configuration classes visible to the application bootstrapping.



If you don't know how to register an Auto-configuration class and bypass this step, your solution will not work.



Spring Boot only prioritizes explicitly registered Auto-configuration classes and not nested classes `@Configuration` classes within them.

- 6. Augment the Application module pom to address dependencies
 - a. Add a dependency on the Starter Module
 - b. Create a profile (`autos`) that adds a direct dependency on the AutoRentals implementation module. The "assignment starter" provides an example of this.
 - c. Create a profile (`tools`) that adds a direct dependency on the ToolRentals implementation module.
- 7. Verify your solution will determine its results based on the available classes and properties at runtime. Your solution must have the following behavior
 - a. no Maven profiles active and no properties provided

```
$ mvn dependency:list -f *-autoconfig-app | egrep 'ejava-student.*module'  
(starter module)  
(interface module)  
①
```

```
$ mvn clean package
$ java -jar *-autoconfig-app/target/*-autoconfig-app-*-bootexec.jar

rentals.active=(not supplied)
rentals.preference=(not supplied)
Rentals is not active ②
```

① no RentalsService implementation jars in dependency classpath

② no implementation was injected because none in the classpath

b. **autos** only Maven profile active and no properties provided

```
$ mvn dependency:list -f *-autoconfig-app -P autos | egrep 'ejava-
student.*module'
(starter module)
(interface module)
(AutoRentals implementation module) ①
```

```
$ mvn clean package -P autos
$ java -jar *-autoconfig-app/target/*-autoconfig-app-*-bootexec.jar
```

```
rentals.active=(not supplied)
rentals.preference=(not supplied)
Rentals has started, rental:{autoRental0} ②
```

① AutoRentals implementation JAR in dependency classpath

② AutoRentalsService was injected because only implementation in classpath

c. **tools** only Maven profile active and no properties provided

```
$ mvn dependency:list -f *-autoconfig-app -P tools | egrep 'ejava-
student.*module'
(starter module)
(interface module)
(ToolRentals implementation module) ①
```

```
$ mvn clean package -P tools
$ java -jar *-autoconfig-app/target/*-autoconfig-app-*-bootexec.jar
```

```
rentals.active=(not supplied)
rentals.preference=(not supplied)
Rentals has started, rental:{toolRental0} ②
```

① ToolRentals implementation JAR in dependency classpath

② ToolRentalsService was injected because only implementation in classpath

d. **autos** and **tools** Maven profiles active

```

$ mvn dependency:list -f *-autoconfig-app -P tools,autos | egrep 'ejava-
student.*module'
(starter module)
(interface module)
(AutoRentals implementation module) ①
(ToolRentals implementation module) ②

$ mvn clean install -P autos,autos
$ java -jar *-autoconfig-app/target/*-autoconfig-app-*-bootexec.jar

rentals.active=(not supplied)
rentals.preference=(not supplied)
Rentals has started, rental:{autoRental0} ③

```

- ① AutoRentals implementation JAR in dependency classpath
- ② ToolRentals implementation JAR in dependency classpath
- ③ AutoRentalsService was injected because of higher-priority

e. **autos** and **tools** Maven profiles active and Spring property **Rental.preference=tools**

```

$ mvn clean install -P autos,autos ①
java -jar rentals-autoconfig-app/target/rentals-autoconfig-app-1.0-SNAPSHOT-
bootexec.jar --rentals.preference=autos ②

rentals.active=(not supplied)
rentals.preference=autos
Rentals has started, rental:{toolRental0} ③

```

- ① AutoRental and ToolRental implementation JARs in dependency classpath
- ② **rentals.preference** property supplied with **autos** value
- ③ ToolRentalsService implementation was injected because of preference specified

f. **autos** and **tools** Maven profiles active and Spring property **rentals.active=false**

```

$ mvn clean install -P autos,autos ①

$ java -jar rentals-autoconfig-app/target/rentals-autoconfig-app-1.0-SNAPSHOT-
bootexec.jar --rentals.active=false ②

rentals.active=false
rentals.preference=(not supplied)
Rentals is not active ③

```

- ① AutoRental and ToolRental implementation JARs in dependency classpath
- ② **rentals.active** property supplied with **false** value

- ③ no implementation was injected because feature deactivated with property value
8. Turn in a source tree with a complete Maven module that will build and demonstrate the Auto-Configuration property processing and output of this application.

122.4.4. Grading

Your solution will be evaluated on:

1. Create a `@Configuration` class/`@Bean` factory method to be registered based on the result of a condition at startup
 - a. whether your solution provides the intended implementation class based on the runtime environment
2. Create a Spring Boot Auto-configuration module to use as a "Starter"
 - a. whether you have successfully packaged your `@Configuration` classes as Auto-Configuration classes outside the package scanning of the `@SpringBootApplication`
3. Bootstrap Auto-configuration classes into applications using a `AutoConfiguration.imports` metadata file
 - a. whether you have bootstrapped your Auto-Configuration classes so they are processed by Spring Boot at application startup
4. Create a conditional component based on the presence of a property value
 - a. whether you activate or deactivate a `@Bean` factory based on the presence or absence of a specific the a specific property
5. Create a conditional component based on a missing component
 - a. whether you activate or deactivate a `@Bean` factory based on the presence or absence of a specific `@Component`
6. Create a conditional component based on the presence of a class
 - a. whether you activate or deactivate a `@Bean` factory based on the presence or absence of a class
 - b. whether your starter causes unnecessary dependencies on the Application module
7. Define a processing dependency order for Auto-configuration classes
 - a. whether your solution is capable of implementing the stated priorities of which bean implementation to instantiate under which conditions

122.4.5. Additional Details

1. A starter project is available in `autorentals-starter/assignment1-autorentals-autoconfig`. Modify Maven groupId and Java package if used.
2. A unit integration test is supplied to check the results. We will cover testing very soon. Activate the test when you are ready to get feedback results. The test requires:
 - All classes be below the `info.ejava_student` Java package
 - The component class injected with the dependency have the bean identity of `appCommand`.

- The injected service made available via `getRentalsService()` method within `appCommand`.

Chapter 123. Assignment 1b: Logging

123.1. Application Logging

123.1.1. Purpose

In this portion of the assignment, you will demonstrate your knowledge of injecting and calling a logging framework. You will:

1. obtain access to an SLF4J Logger
2. issue log events at different severity levels
3. format log events for regular parameters
4. filter log events based on source and severity thresholds

123.1.2. Overview

In this portion of the assignment, you are going to implement a call thread through a set of components that are in different Java packages that represent at different levels of the architecture. Each of these components will setup an SLF4J **Logger** and issue logging statements relative to the thread.

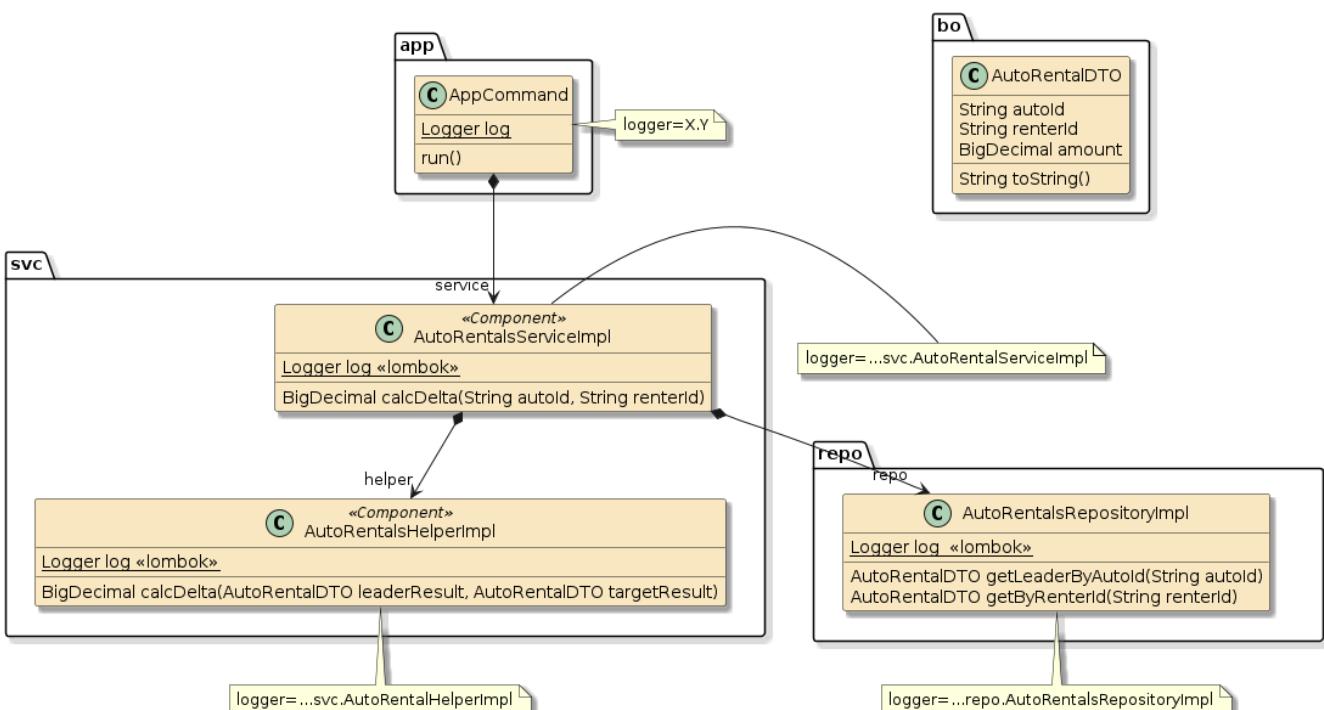


Figure 33. Application Logging

123.1.3. Requirements



All data is fake and random here. The real emphasis should be placed on the logging events that occur on the different loggers and not on creating a realistic AutoRental result.

1. Create several components in different Java sub-packages (app, svc, and repo)
 - a. an `AppCommand` component class in the `app` Java sub-package
 - b. a `AutoRentalsServiceImpl` component class in the `svc` Java sub-package
 - c. a `AutoRentalsHelperImpl` component class in the `svc` Java sub-package
 - d. a `AutoRentalsRepositoryImpl` component class in the `repo` Java sub-package
2. Implement a chain of calls from the `AppCommand @Component run()` method through the other components.

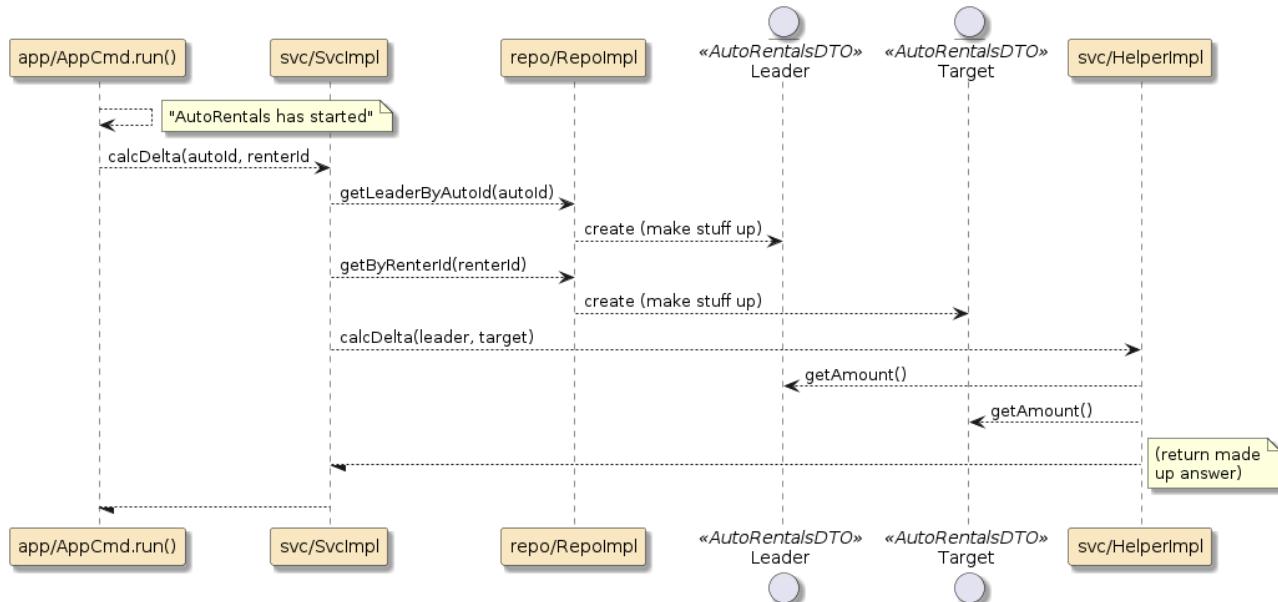


Figure 34. Required Call Sequence

- a. `AppCommand.run()` logs message with text "AutoRentals has started"
- b. `AppCommand.run()` calls `ServiceImpl.calcDelta(autoId, renterId)` with a `autoId` and `renterId` to determine a delta between the two instances
- c. `ServiceImpl.calcDelta(autoId, renterId)` calls `RepositoryImpl` (`getLeaderByAutoId(autoId)` and `getByRenterId(renterId)`) to get `AutoRentalDTOs`
 - i. `RepositoryImpl` can create transient instances with provided Ids and fake remaining properties
- d. `ServiceImpl.calcDelta(autoId, renterId)` also calls `ResultsHelper.calcDelta()` to get a delta between the two `AutoRentalDTOs`
- e. `HelperImpl.calcDelta(leader, target)` calls `AutoRentalDTO.getAmount()` on the two provided `AutoRentalDTO` instances to determine the delta



The focus of logging and this assignment really starts at this point and forward. I have considered writing the interaction logic above for you to eliminate this distraction within a logging assignment. However, I found that at this early point in the semester and assignments — this is also a good time to practice essential skills of creating components with dependencies and simple interactions.

3. Implement a `toString()` method in `AutoRentalDTO` that includes the `autoId`, `renterId`, and `amount` information.
4. Instantiate an SLF4J `Logger` into each of the four components
 - a. manually instantiate a static final `Logger` with the name "X.Y" in `AppCommand`
 - b. leverage the Lombok library to instantiate a `Logger` with a name based on the Java package and name of the hosting class for all other components
5. Implement logging statements in each of the methods
 - a. the severity of `RepositoryImpl` logging events are all TRACE
 - b. the severity of `HelperImpl.calcDelta()` logging events are DEBUG and TRACE (there must be at least two — one of each and no other levels)
 - c. the severity of `ServiceImpl.calcDelta()` logging events are all INFO and TRACE (there must be at least two — one of each and no other levels)
 - d. the severity of `AppCommand` logging events are all INFO (and no other levels)
6. Output available results information in log statements
 - a. Leverage the SLF4J parameter formatting syntax when implementing the log
 - b. For each of the INFO and DEBUG statements, include only the `AutoRentalDTO` property values (e.g., `autoId`, `renterId`, `timeDelta`)



Use direct calls on individual properties for INFO and DEBUG statements

i.e., `autoRental.getAutoId()`, `autoRental.getRenterId()`, etc.

- c. For each of the TRACE statements, use the inferred `AutoRentalDTO.toString()` method to log the `AutoRentalDTO`.



Use inferred `toString()` on passed object on TRACE statements

i.e., `log.debug("...", autoRental)` — no direct calls to `toString()`

7. Supply two profiles
 - a. the root logger must be turned off by default (e.g., in `application.properties`)
 - b. an `app-debug` profile that turns on DEBUG and above priority (e.g., DEBUG, INFO, WARN, ERROR) logging events for all loggers in the application, including "X.Y"
 - c. a `repo-only` profile that turns on only log statements from the repo class(es).
8. Wrap your solution in a Maven build that executes the JAR three times with:
 - a. (no profile) - no logs should be produced
 - b. `app-debug` profile
 - i. DEBUG and higher priority logging events from the application (including "X.Y") are output to console
 - ii. no TRACE messages are output to the console
 - c. `repo-only` profile

- i. logging events from repository class(es) are output to the console
- ii. no other logging events are output to the console

123.1.4. Grading

Your solution will be evaluated on:

1. obtain access to an SLF4J Logger
 - a. whether you manually instantiated a Logger into the AppCommand `@Component`
 - b. whether you leveraged Lombok to instantiate a Logger into the other `@Components`
 - c. whether your App Command `@Component` logger was named "X.Y"
 - d. whether your other `@Component` loggers were named after the package/class they were declared in
2. issue log events at different severity levels
 - a. where logging statements issued at the specified verbosity levels
3. format log events for regular parameters
 - a. whether SLF4J format statements were used when including variable information
4. filter log events based on source and severity thresholds
 - a. whether your profiles set the logging levels appropriately to only output the requested logging events

123.1.5. Other Details

1. You may use any means to instantiate/inject the components (i.e., `@Bean` factories or `@Component` annotations)
2. You are encouraged to use Lombok to declare constructors, getter/setter methods, and anything else helpful **except for the manual instantiation of the "X.Y" logger in `AppCommand`.**
3. A starter project is available in `autorentals-starter/assignment1-autorentals-logging`. It contains a Maven pom that is configured to build and run the application with the following profiles for this assignment:
 - no profile
 - `app-debug`
 - `repo-only`
 - `appenders` (used later in this assignment)
 - `appenders` and `trace`
4. There is an integration unit test (`MyLoggingNTest`) provided in the starter module. We will discuss testing very soon. Enable this test when you are ready to have the results evaluated.

123.2. Logging Efficiency

123.2.1. Purpose

In this portion of the assignment, you will demonstrate your knowledge of making suppressed logging efficient. You will:

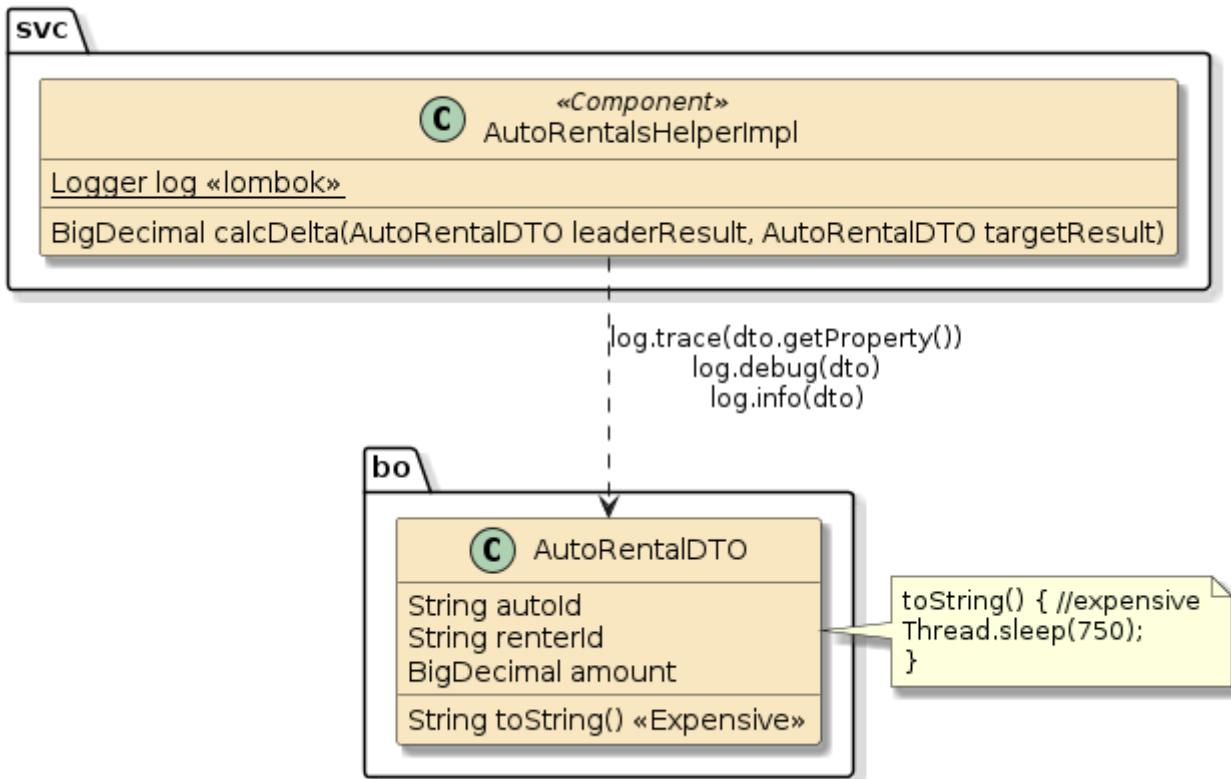
1. efficiently bypass log statements that do not meet criteria

123.2.2. Overview

In this portion of the assignment, you are going to increase the cost of calling `toString()` on the logged object and work to only pay that penalty when needed.



Make your changes to the previous logging assignment solution. Do not create a separate module for this work.



123.2.3. Requirements

1. Update the `toString()` method in `AutoRentalDTO` to be expensive to call
 - a. artificially insert a 750 milliseconds delay within the `toString()` call
2. Refactor your log statements, if required, to only call `toString()` when TRACE is active
 - a. leverage the SLF4J API calls to make that as simple as possible

123.2.4. Grading

Your solution will be evaluated on:

1. efficiently bypass log statements that do not meet criteria
 - a. whether your `toString()` method paused the calling thread for 750 milliseconds when TRACE threshold is activated
 - b. whether the calls to `toString()` are bypassed when priority threshold is set higher than TRACE
 - c. the simplicity of your solution

123.2.5. Other Details

1. Include these modifications with the previous work on this overall logging assignment. Meaning—there will not be a separate module turned in for this portion of the assignment.
2. The `app-debug` should not exhibit any additional delays. The `repo-only` should exhibit a 1.5 (2x750msec) second delay.
3. There is an integration unit test ([MyLoggingEfficiencyNTest](#)) provided in the starter module. We will discuss testing very soon. Enable this test when you are ready to have the results evaluated.

123.3. Appenders and Custom Log Patterns

123.3.1. Purpose

In this portion of the assignment, you will demonstrate your knowledge of assigning appenders to loggers and customizing logged events. You will:

1. filter log events based on source and severity thresholds
2. customize log patterns
3. customize appenders
4. add contextual information to log events using Mapped Diagnostic Context
5. use Spring Profiles to conditionally configure logging

123.3.2. Overview

In this portion of the assignment you will be creating/configuring a few basic appenders and mapping loggers to them—to control what, where, and how information is logged. This will involve profiles, property files, and a logback configuration file.



Make your changes to the original logging assignment solution. Do not create a separate module for this work.

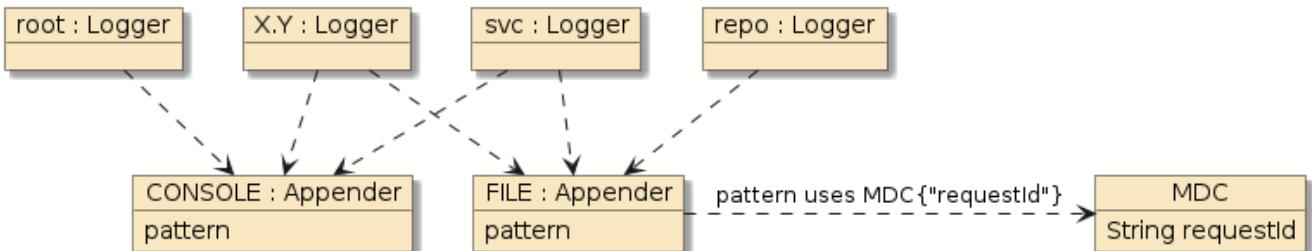


Figure 35. Appenders and Custom Log Patterns



Except for setting the MDC, you are writing no additional code in this portion of the assignment. Most of your work will be in filling out the logback configuration file and setting properties in profile-based property files to tune logged output.

123.3.3. Requirements

1. Declare two Appenders as part of a custom Logback configuration
 - a. CONSOLE appender to output to stdout
 - b. FILE appender to output to a file `target/logs/appenders.log`
2. Assign the Appenders to Loggers
 - a. root logger events **must** be assigned to the CONSOLE Appender
 - b. any log events issued to the "X.Y" Logger must be assigned to **both** the CONSOLE and FILE Appenders
 - c. any log events issued to the "...svc" Logger must also be assigned to **both** the CONSOLE and FILE Appenders
 - d. any log events issued to the "...repo" Logger must **only** be assigned to the FILE Appender



Remember "additivity" rules for inheritance and appending assignment



These are the only settings you need to make within the Appender file. All other changes can be done through properties. However, there will be no penalty (just complexity) in implementing other mechanisms.

3. Add an `appenders` profile that
 - a. automatically enacts the requirements above
 - b. sets a base of INFO severity and up for all loggers with your application
4. Add a `requestId` property to the Mapped Diagnostic Context (MDC)

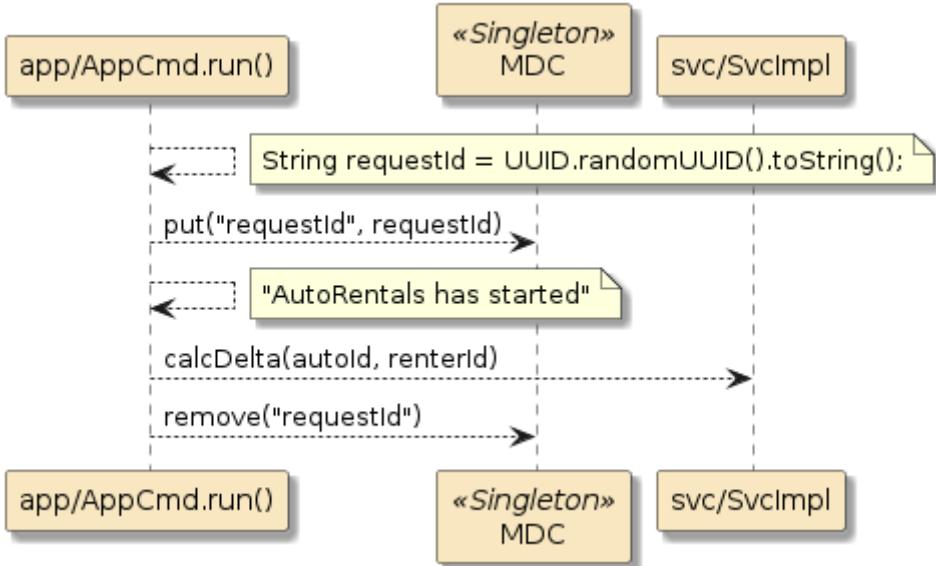


Figure 36. Initialize Mapped Diagnostic Context (MDC)

- generate a random/changing value using a 36 character UUID String



Example: `UUID.randomUUID().toString()` ⇒ `d587d04c-9047-4aa2-bfb3-82b25524ce12`

- insert the value prior to the first logging statement — in the `AppCommand` component
 - remove the MDC property when processing is complete within the `AppCommand` component
- Declare a custom logging pattern in the `appenders` profile that includes the MDC `requestId` value in each log statements written by the FILE Appender
 - The MDC `requestId` is only output by the FILE Appender. Encase the UUID within square brackets so that it can be found in a pattern search more easily



Example: `[d587d04c-9047-4aa2-bfb3-82b25524ce12]`

- The MDC `requestId` is not output by the CONSOLE Appender
- Add an additional `trace` profile that
 - activates logging events at TRACE severity and up for all loggers with your application
 - adds method and line number information to all entries in the FILE Appender but not the CONSOLE Appender. Use a format of `method:lineNumber` in the output.



Example: `run:27`



Optional: Try defining the logging pattern once with an optional property variable that can be used to add method and line number expression versus repeating the definition twice.

- Apply the `appenders` profile to

- output logging events at INFO severity and up to both CONSOLE and FILE Appenders

- b. include the MDC `requestId` in events logged by the FILE Appender
 - c. not include method and line number information in events logged
8. Apply the `appenders` and `trace` profiles to
- a. output logging events at TRACE severity and up to both CONSOLE and FILE Appenders
 - b. continue to include the MDC `requestId` in events logged by the FILE Appender
 - c. add method and line number information in events logged by the FILE Appender

123.3.4. Grading

Your solution will be evaluated on:

1. filter log events based on source and severity thresholds
 - a. whether your log events from the different Loggers were written to the required appenders
 - b. whether a log event instance appeared at most once per appender
2. customize log patterns
 - a. whether your FILE Appender output was augmented with the `requestId` when `appenders` profile was active
 - b. whether your FILE Appender output was augmented with method and line number information when `trace` profile was active
3. customize appenders
 - a. whether a FILE and CONSOLE appender were defined
 - b. whether a custom logging pattern was successfully defined for the FILE Logger
4. add contextual information to log events using Mapped Diagnostic Context
 - a. whether a `requestId` was added to the Mapped Data Context (MDC)
 - b. whether the `requestId` was included in the customized logging pattern for the FILE Appender when the `appenders` profile was active
5. use Spring Profiles to conditionally configure logging
 - a. whether your required logging configurations were put in place when activating the `appenders` profile
 - b. whether your required logging configurations were put in place when activating the `appenders` and `trace` profiles

123.3.5. Other Details

1. You may use the default Spring Boot LogBack definition for the FILE and CONSOLE Appenders (i.e., include them in your logback configuration definition).

Included Default Spring Boot LogBack definitions

```
<configuration>
  <include resource="org/springframework/boot/logging/logback/defaults.xml"/>
```

```

<include resource="org/springframework/boot/logging/logback/console-
appender.xml"/>
<include resource="org/springframework/boot/logging/logback/file-appender.xml
"/>
...

```

2. Your `appenders` and `trace` profiles may re-define the logging pattern for the FILE logger or add/adjust parameterized definitions. However, try to implement an optional parameterization as your first choice to keep from repeating the same definition.
3. The following snippet shows an example resulting logfile from when `appenders` and then `appenders,trace` profiles were activated. Yours may look similar to the following:

Example target/logs/appenders.log - "appenders" profile active

```

$ rm target/logs/appenders.log
$ java -jar target/assignment1-*-logging-1.0-SNAPSHOT-bootexec.jar
--spring.profiles.active=appenders
$ head target/logs/appenders.log

head target/logs/appenders.log          ①
21:46:01.335  INFO -- [c934e045-1294-43c9-8d22-891eec2b8b84]  Y : AutoRentals has
started ②

```

① `requestId` is supplied in all FILE output when `appenders` profile active

② no method and line number info supplied

Example target/logs/appenders.log - "appenders,trace" profiles active

```

$ rm target/logs/appenders.log
$ java -jar target/assignment1-*-logging-1.0-SNAPSHOT-bootexec.jar
--spring.profiles.active=appenders,trace
$ head target/logs/appenders.log

$ head target/logs/appenders.log          ①
21:47:33.784  INFO -- [0289d00e-5b28-4b01-b1d5-1ef8cf203d5d]  Y.run:27 :
AutoRentals has started ②

```

① `requestId` is supplied in all FILE output when `appenders` profile active

② method and line number info are supplied

4. There is a set of unit integration tests provided in the support module. We will cover testing very soon. Enable them when you are ready to evaluate your results.

Chapter 124. Assignment 1c: Testing

The following parts are broken into different styles of conducting a pure unit test and unit integration test—based on the complexity of the class under test. None of the approaches are deemed to be "the best" for all cases.

- tests that run without a Spring context can run blazingly fast, but lack the target runtime container environment
- tests that use Mocks keep the focus on the subject under test, but don't verify end-to-end integration
- tests that assemble real components provide verification of end-to-end capability but can introduce additional complexities and performance costs

It is important that you come away knowing how to implement the different styles of unit testing so that they can be leveraged based on specific needs.

124.1. Demo

The `assignment1-autorentals-testing` assignment starter contains a `@SpringBootApplication` main class and a some demonstration code that will execute at startup when using the `demo` profile.

Application Demonstration

```
$ mvn package -Pdemo
06:34:21.217 INFO -- RentersServiceImpl : renter added: RenterDTO(id=null,
firstName=warren, lastName=buffet, dob=1930-08-30)
06:34:21.221 INFO -- RentersServiceImpl : invalid renter: RenterDTO(id=null,
firstName=future, lastName=buffet, dob=2023-07-14), [renter.dob: must be greater than
12 years]
```

You can follow that thread of execution through the source code to get better familiarity with the code you will be testing.

Starter Contains 2 Example Ways to Implement CommandLineRunner

There are actually 2 sets of identical output generated during the provided demo execution. The starter is supplied with two example ways to implement a `CommandLineRunner`: as a class and as a lambda function.

Implementing CommandLineRunner as a Class that Implements Interface



```
@Component
@RequiredArgsConstructor
static class Init implements CommandLineRunner {
    private final RentersService rentersService;
    @Override
    public void run(String... args) {
```

```
@Bean  
CommandLineRunner lambdaDemo(RentersService rentersService) {  
    return (args)->{
```

124.2. Unit Testing

124.2.1. Purpose

In this portion of the assignment, you will demonstrate your knowledge of implementing a unit test for a Java class. You will:

1. write a test case and assertions using JUnit 5 "Jupiter" constructs
2. leverage the AssertJ assertion library
3. execute tests using Maven Surefire plugin

124.2.2. Overview

In this portion of the assignment, you are going to implement a test case with 2 unit tests for a completed Java class.

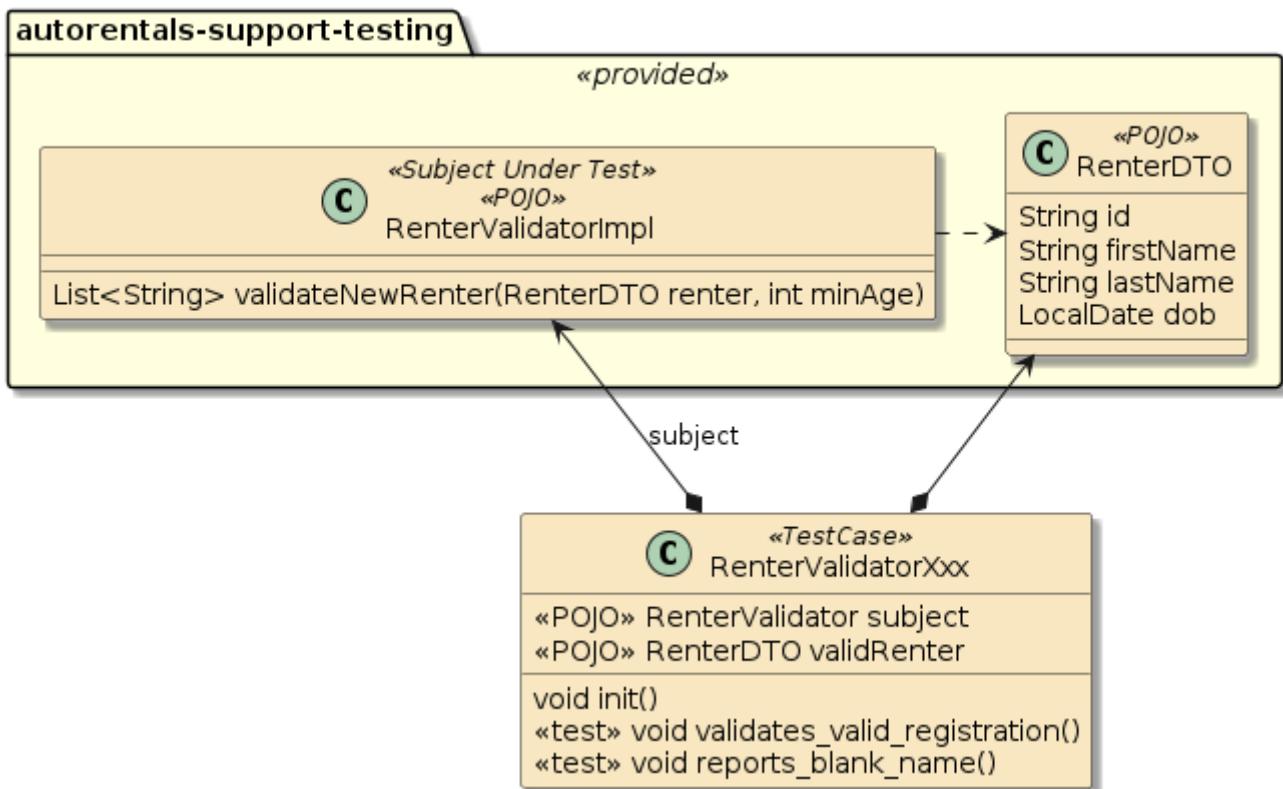


Figure 37. Unit Test

The code under test is 100% complete and provided to you in a separate `autorentals-support-testing` module.

Code Under Test

```
<dependency>
    <groupId>info.ejava.assignments.testing.autorentals</groupId>
    <artifactId>autorentals-support-testing</artifactId>
    <version>${ejava.version}</version>
</dependency>
```

Your assignment will be implemented in a module you create and form a dependency on the implementation code.

124.2.3. Requirements

1. Start with a dependency on supplied and completed `RenterValidatorImpl` and `RenterDTO` classes in the `autorentals-support-testing` module. You only need to understand and test them. You do not need to implement or modify anything being tested.
 - a. `RenterValidatorImpl` implements a `validateNewRenter` method that returns a `List<String>` with identified validation error messages
 - b. `RenterDTO` must have the following to be considered valid for registration:
 - i. null `id`
 - ii. non-blank `firstName` and `lastName`
 - iii. `dob` older than `minAge`
2. Implement a plain unit test case class for `RenterValidatorImpl`
 - a. the test must be implemented **without** the use of a Spring context
 - b. all instance variables for the test case must come from plain POJO calls
 - c. tests must be implemented with JUnit 5 (Jupiter) constructs.
 - d. tests must be implemented using AssertJ assertions. Either BDD or regular form of assertions is acceptable.
3. The unit test case must have an `init()` method configured to execute "before each" test
 - a. this can be used to initialize variables prior to each test
4. The unit test case must have a test that verifies a valid `RenterDTO` will be reported as **valid**.
5. The unit test case must have a test method that verifies an invalid `RenterDTO` will be reported as **invalid** with a string message for each error.
6. Name the test so that it automatically gets executed by the Maven Surefire plugin.

124.2.4. Grading

Your solution will be evaluated on:

1. write a test case and assertions using JUnit 5 "Jupiter" constructs
 - a. whether you have implemented a pure unit test absent of any Spring context
 - b. whether you have used JUnit 5 versus JUnit 4 constructs

- c. whether your `init()` method was configured to be automatically called "before each" test
 - d. whether you have tested with a valid and invalid `RenterDTO` and verified results where appropriate
2. leverage AssertJ assertion libraries
 - a. whether you have used assertions to identify pass/fail
 - b. whether you have used the AssertJ assertions
 3. execute tests using Maven Surefire plugin
 - a. whether your unit test is executed by Maven surefire during a build

124.2.5. Additional Details

1. A quick start project is available in [autorentals-starter/assignment1-autorentals-testing](#), but
 - a. copy the module into your own area
 - b. modify at least the Maven groupId and Java package when used
2. You are expected to form a dependency on the [autorentals-support-testing](#) module. The only things present in your `src/main` would be demonstration code that is supplied to you in the starter—but not part of any requirement.

124.3. Mocks

124.3.1. Purpose

In this portion of the assignment, you will demonstrate your knowledge of instantiating a Mock as part of unit testing. You will:

1. implement a mock (using Mockito) into a JUnit unit test
2. define custom behavior for a mock
3. capture and inspect calls made to mocks by subjects under test

124.3.2. Overview

In this portion of the assignment, you are going to again implement a unit test case for a class and use a mock for one of its dependencies.

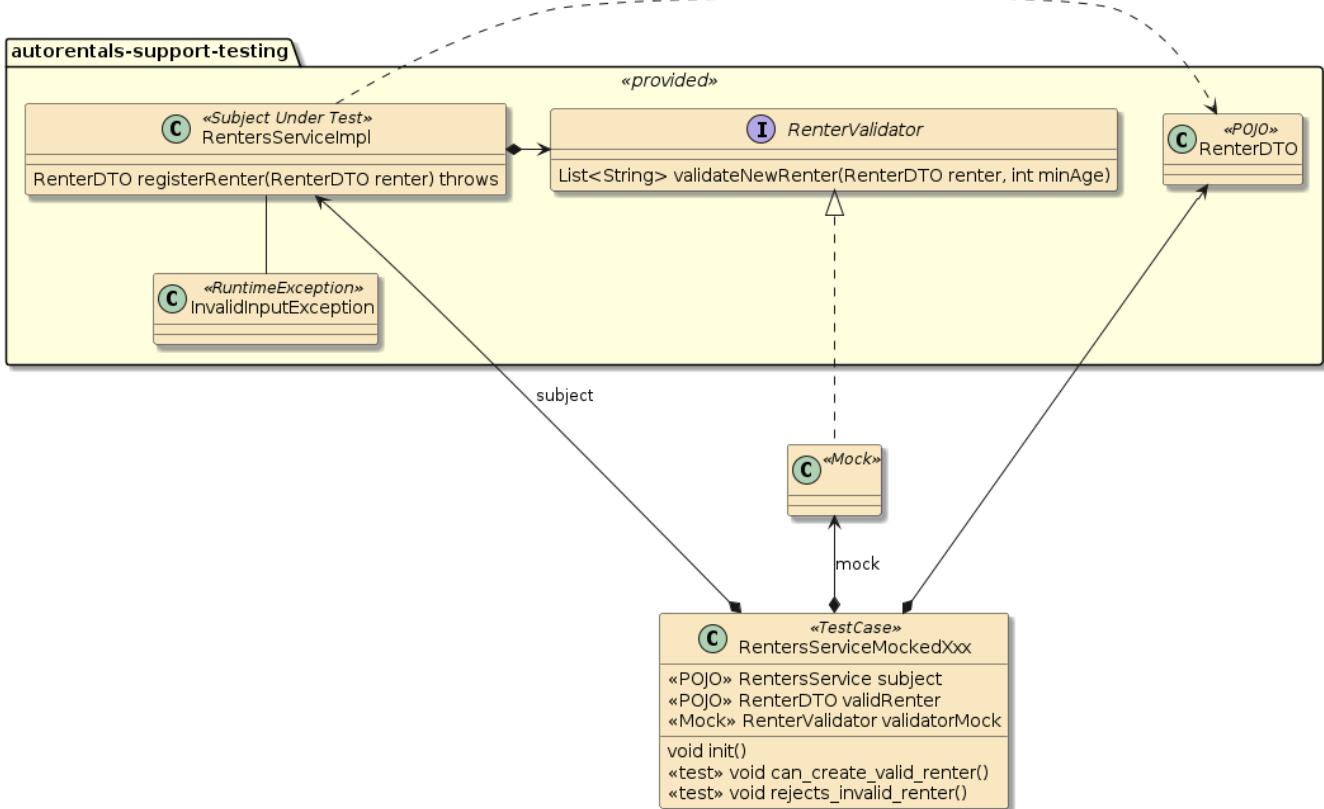


Figure 38. Unit Testing with Mocks

124.3.3. Requirements

1. Start with a dependency on supplied and completed `RentersServiceImpl` and other classes in the `autorentals-support-testing` module. You only need to understand and test them. You do not need to implement or modify anything being tested.
 - a. `RentersServiceImpl` implements a `createRenter` method that
 - i. validates the renter using a `RenterValidator` instance
 - ii. assigns the `id` if valid
 - iii. throws an exception with the error messages from the validator if invalid
2. Implement a unit test case for the `RentersService` to verify validation for a valid and invalid `RenterDTO`
 - a. the test case must be implemented without the use of a Spring context
 - b. all instance variables for the test case, except for the mock, must come from plain POJO calls
 - c. tests must be implemented using AssertJ assertions. Either BDD or regular form of assertions is acceptable.
 - d. a Mockito Mock must be used for the `RenterValidator` instance. You may **not** use the `RenterValidatorImpl` class as part of this test
3. The unit test case must have an `init()` method configured to run "before each" test and initialize the `RentersServiceImpl` with the Mock instance for `RenterValidator`.
4. The unit test case must have a test that verifies a valid registration will be handled as **valid**.
 - a. configure the Mock to return an empty `List<String>` when asked to validate the renter.



Understand how the default Mock behaves before going too far with this.

- b. programmatically verify the Mock was called to validate the `RenterDTO` as part of the test criteria
5. The unit test case must have a test method that verifies an **invalid** registration will be reported with an exception.
 - a. configure the Mock to return a `List<String>` with errors for the renter
 - b. programmatically verify the Mock was called 1 time to validate the `RenterDTO` as part of the test criteria
6. Name the test so that it automatically gets executed by the Maven Surefire plugin.

This assignment is not to test the Mock. It is a test of the Subject using a Mock



You are not testing or demonstrating the Mock. Assume the Mock works and use the capabilities of the Mock to test the subject(s) they are injected into. Place any experiments with the Mock in a separate Test Case and keep this assignment focused on testing the subject (with the functioning Mock).

124.3.4. Grading

Your solution will be evaluated on:

1. implement a mock (using Mockito) into a JUnit unit test
 - a. whether you used a Mock to implement the `RenterValidator` as part of this unit test
 - b. whether you used a Mockito Mock
 - c. whether your unit test is executed by Maven surefire during a build
2. define custom behavior for a mock
 - a. whether you successfully configured the Mock to return an empty collection for the valid renter
 - b. whether you successfully configured the Mock to return a collection of error messages for the invalid renter
3. capture and inspect calls made to mocks by subjects under test
 - a. whether you programmatically checked that the Mock validation method was called as a part of registration using Mockito library calls

124.3.5. Additional Details

1. This portion of the assignment is expected to primarily consist of one additional test case added to the `src/test` tree.
2. You may use BDD or non-BDD syntax for this test case and tests.

124.4. Mocked Unit Integration Test

124.4.1. Purpose

In this portion of the assignment, you will demonstrate your knowledge of implementing a unit integration test using a Spring context and Mock beans. You will:

1. to implement unit integration tests within Spring Boot
2. implement mocks (using Mockito) into a Spring context for use with unit integration tests

124.4.2. Overview

In this portion of the assignment, you are going to implement an injected Mock bean that will be injected by Spring into both the `RentersServiceImpl @Component` for operational functionality and the unit integration test for configuration and inspection commands.

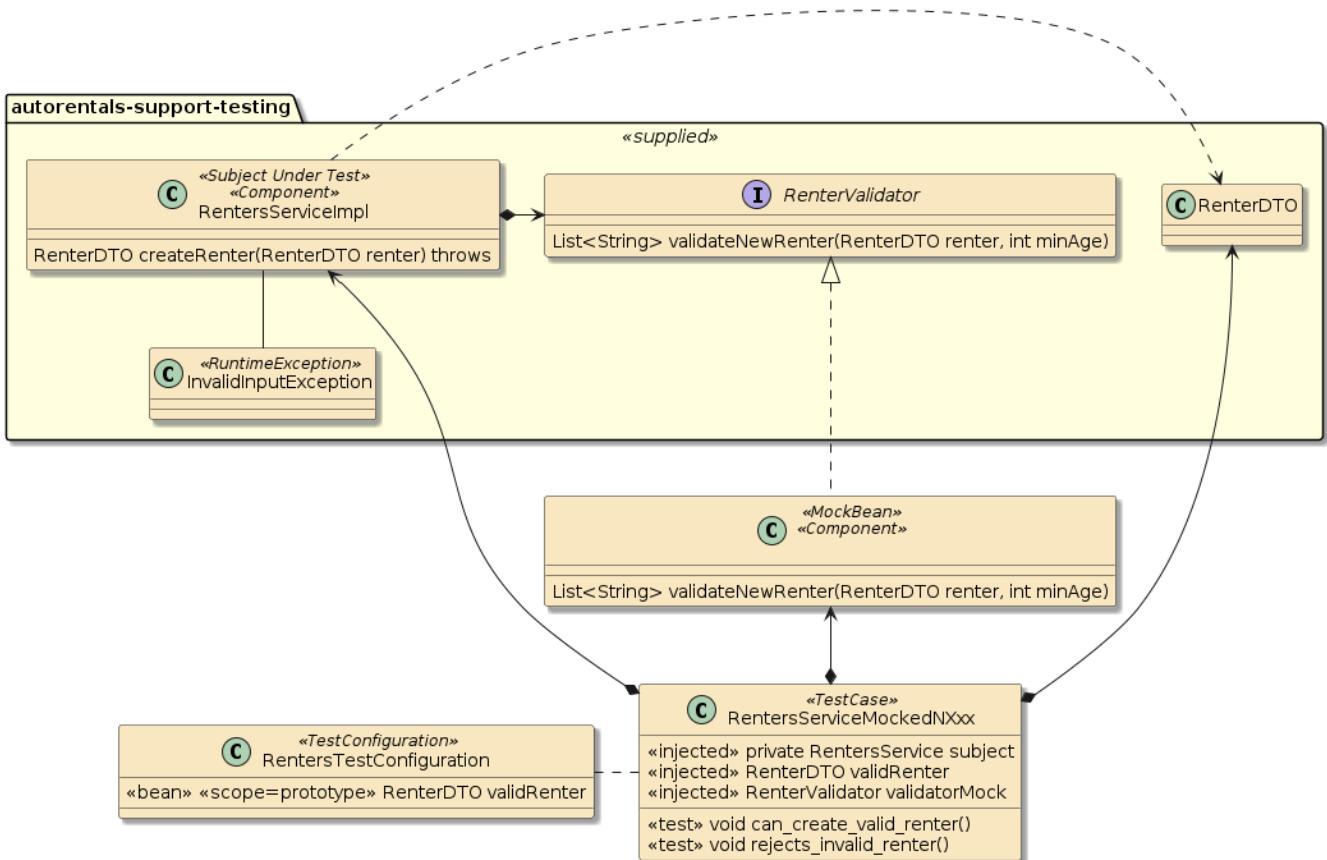


Figure 39. Mocked Unit Integration Test

124.4.3. Requirements

1. Start with a supplied, completed, and injectable 'RentersServiceImpl' versus instantiating one like you did in the pure unit tests.
2. Implement a unit integration test for the `RentersService` for a valid and invalid `RenterDTO`
 - a. the test must be implemented using a Spring context
 - b. all instance variables for the test case must come from injected components — even trivial

ones.

- c. the `RenterValidator` must be implemented as Mockito Mock/Spring bean and injected into both the `RenterValidatorImpl @Component` and accessible in the unit integration test. You may **not** use the `RenterValidatorImpl` class as part of this test.
 - d. define and inject a `RenterDTO` for a valid renter as an example of a bean that is unique to the test. This can come from a `@Bean` factory
3. The unit integration test case must have a test that verifies a **valid** registration will be handled as valid.
 4. The unit integration test case must have a test method that verifies an **invalid** registration will be reported with an exception.
 5. Name the unit integration test so that it automatically gets executed by the Maven Surefire plugin.

124.4.4. Grading

Your solution will be evaluated on:

1. to implement unit integration tests within Spring Boot
 - a. whether you implemented a test case that instantiated a Spring context
 - b. whether the subject(s) and their dependencies were injected by the Spring context
 - c. whether the test case verified the requirements for a valid and invalid input
 - d. whether your unit test is executed by Maven surefire during a build
2. implement mocks (using Mockito) into a Spring context for use with unit integration tests
 - a. whether you successfully declared a Mock bean that was injected into the necessary components under test and the test case for configuration

124.4.5. Additional Details

1. This portion of the assignment is expected to primarily consist of
 - a. adding one additional test case added to the `src/test` tree
 - b. adding any supporting `@TestConfiguration` or other artifacts required to define the Spring context for the test
 - c. changing the Mock to work with the Spring context
2. Anything you may have been tempted to simply instantiate as `private X x = new X();` can be changed to an injection by adding a `@(Test)Configuration/@Bean` factory to support testing. The point of having the 100% injection requirement is to encourage encapsulation and reuse among Test Cases for all types of test support objects.
3. You may add the `RentersTestConfiguration` to the Spring context using either of the two annotation properties
 - a. `@SpringBootTest.classes`
 - b. `@Import.value`

4. You may want to experiment with applying `@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)` versus the default `@Scope(ConfigurableBeanFactory.SCOPE_SINGLETON)` to your injected Renter and generate a random name in the `@Bean` factory. Every injected `SCOPE_SINGLETON` (default) gets the same instance. `SCOPE_PROTOTYPE` gets a separate instance. Useful to know, but not a graded part of the assignment.

124.5. Unmocked/BDD Unit Integration Testing

124.5.1. Purpose

In this portion of the assignment, you will demonstrate your knowledge of conducting an end-to-end unit integration test that is completely integrated with the Spring context and using Behavior Driven Design (BDD) syntax. You will:

1. make use of BDD acceptance test keywords

124.5.2. Overview

In this portion of the assignment, you are going to implement an end-to-end unit integration test case for two classes integrated/injected using the Spring context with the syntactic assistance of BDD-style naming.

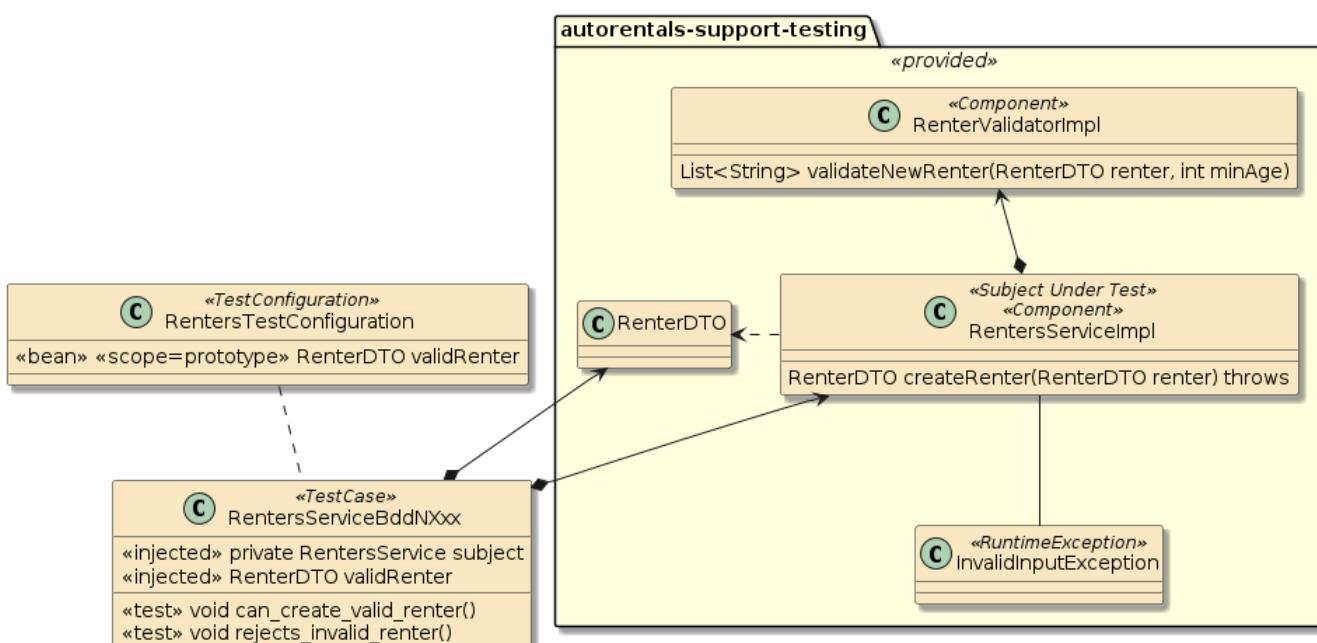


Figure 40. Unmocked/BDD Integration Testing

124.5.3. Requirements

1. Start with a supplied, completed, and injectable `RentersServiceImpl` by creating a dependency on the `autorentals-support-testing` module. There are to be no POJO or Mock implementations of any classes under test.
2. Implement a unit integration test for the `RentersService` for a valid and invalid `RenterDTO`

- a. the test must be implemented using a Spring context
 - b. all instance variables for the test case must come from injected components
 - c. the `RenterValidator` must be injected into the `RentersServiceImpl` using the Spring context. Your test case will not need access to that component.
 - d. define and inject a `RenterDTO` for a valid renter as an example of a bean that is unique to the test. This can come from a `@Bean` factory from a Test Configuration
3. The unit integration test case must have
- a. a display name defined for this test case that includes spaces
 - b. a display name generation policy for contained test methods that includes spaces
4. The unit integration test case must have a test that verifies a valid registration will be handled as **valid**.
- a. use BDD (`then()`) alternative syntax for AssertJ assertions
5. The unit integration test case must have a test method that verifies an **invalid** registration will be reported with an exception.
- a. use BDD (`then()`) alternative syntax for AssertJ assertions
6. Name the unit integration test so that it automatically gets executed by the Maven Surefire plugin.

124.5.4. Grading

Your solution will be evaluated on:

1. make use of BDD acceptance test keywords
 - a. whether you provided a custom display name for the test case that included spaces
 - b. whether you provided a custom test method naming policy that included spaces
 - c. whether you used BDD syntax for AssertJ assertions
2. whether components are injected from Spring context into test
3. whether this test is absent of any Mocks

124.5.5. Additional Details

1. This portion of the assignment is expected to primarily consist of adding a test case that
 - a. is based on the Mocked Unit Integration Test solution, which relies primarily on the beans of the Spring context
 - b. removes any Mocks
 - c. defines a names and naming policies for JUnit
 - d. changes AssertJ syntax to BDD form
2. The "custom test method naming policy" can be set using either an `@Annotation` or `property`. The properties approach has the advantage of being global to all tests within the module.

HTTP API

copyright © 2024 jim stafford (jim.stafford@jhu.edu)

Chapter 125. Introduction

125.1. Goals

The student will learn:

- how the WWW defined an information system capable of implementing system APIs
- identify key differences between a truly RESTful API and REST-like or HTTP-based APIs
- how systems and some actions are broken down into resources
- how web interactions are targeted at resources
- standard HTTP methods and the importance to use them as intended against resources
- individual method safety requirements
- value in creating idempotent methods
- standard HTTP response codes and response code families to respond in specific circumstances

125.2. Objectives

At the conclusion of this lecture and related exercises, the student will be able to:

1. identify API maturity according to the Richardson Maturity Model (RMM)
2. identify resources
3. define a URI for a resource
4. define the proper method for a call against a resource
5. identify safe and unsafe method behavior
6. identify appropriate response code family and value to use in certain circumstances

Chapter 126. World Wide Web (WWW)

The **World Wide Web (WWW)** is an information system of web resources identified by **Uniform Resource Locators (URLs)** that can be interlinked via **hypertext** and transferred using **Hypertext Transfer Protocol (HTTP)**. ^[1] **Web resources** started out being documents to be created, downloaded, replaced, and removed but has progressed to being any identifiable thing—whether it be the entity (e.g., person), something related to that entity (e.g., photo), or an action (e.g., change of address). ^[2]

126.1. Example WWW Information System

The example information system below is of a standard set of content types, accessed through a standard set of methods, and related through location-independent links using URLs.

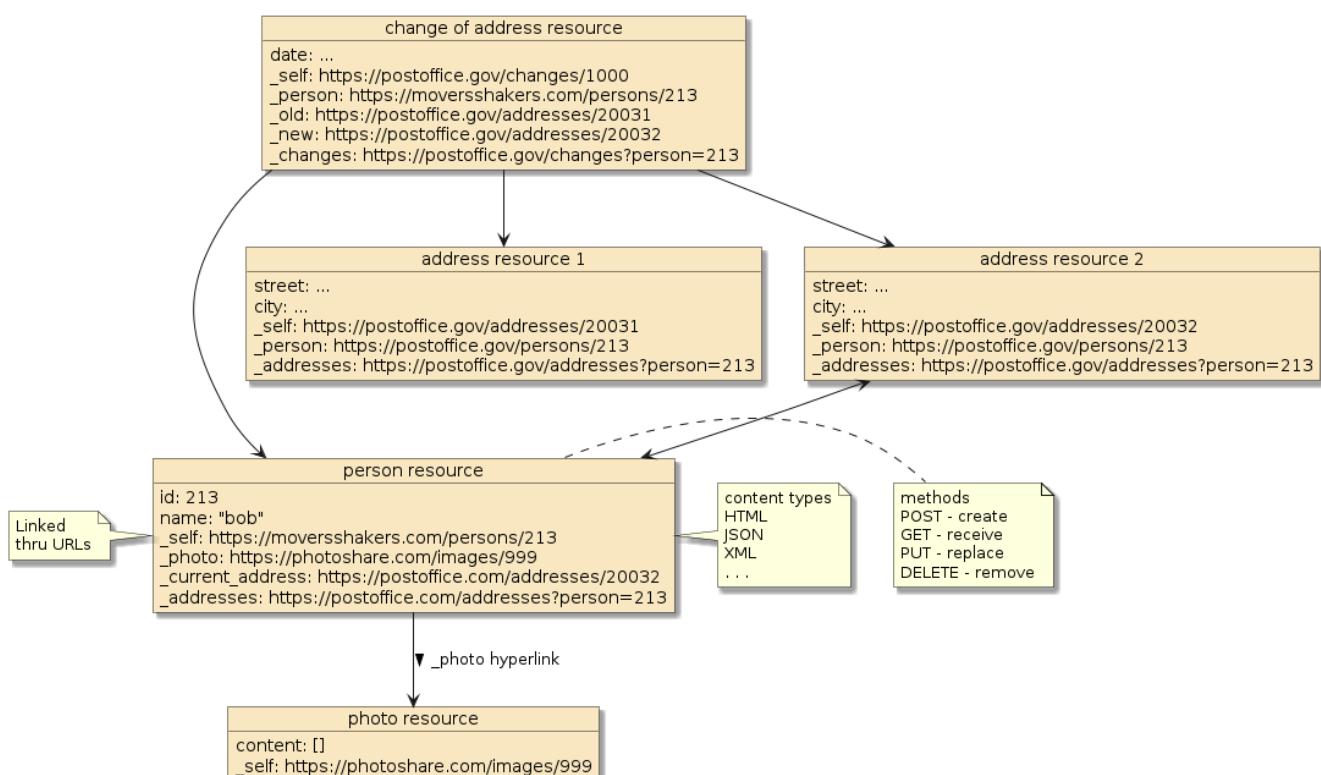


Figure 41. WWW Links Resources thru URLs

[1] ["World Wide Web Wikipedia Page"](#)

[2] ["Web Resource Wikipedia Page"](#)

Chapter 127. REST

Representational State Transfer (REST) is an architectural style for creating web services and web services that conform to this style are considered "Restful" web services [1]. REST was defined in 2000 by Roy Fielding in his [doctoral dissertation](#) that was also used to design HTTP 1.1. [2] REST relies heavily on the concepts and implementations used in the World Wide Web — which centers around web resources addressable using URIs.

127.1. HATEOAS

At the heart of REST is the notion of hyperlinks to represent state. For example, the presence of a `address_change` link may mean the address of a person can be changed and the client accessing that person representation is authorized to initiate the change. The presence of `current_address` and `addresses` links identifies how the client can obtain the current and past addresses for the person. This is shallow description of what is defined as "[Hypermedia As The Engine Of Application State](#)" (**HATEOAS**).

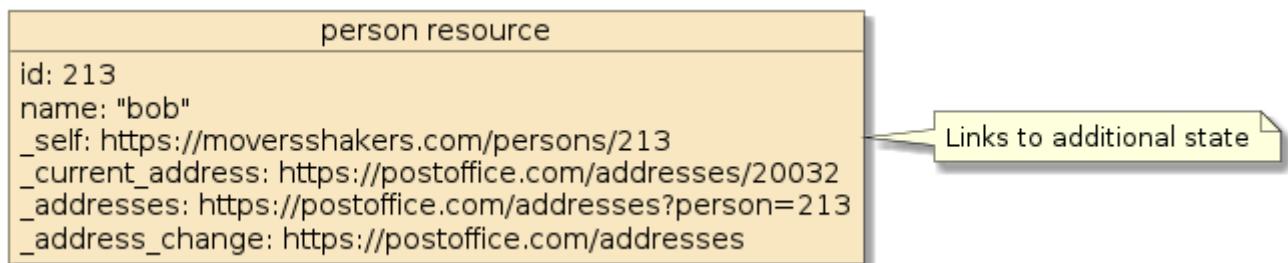


Figure 42. Example of State Represented through Hyperlinks

The interface contract allows clients to dynamically determine current capabilities of a resource and the resource to add capabilities over time.

127.2. Clients Dynamically Discover State

HATEOAS permits the capabilities of client and server to advance independently through the dynamic discovery of links. [3]

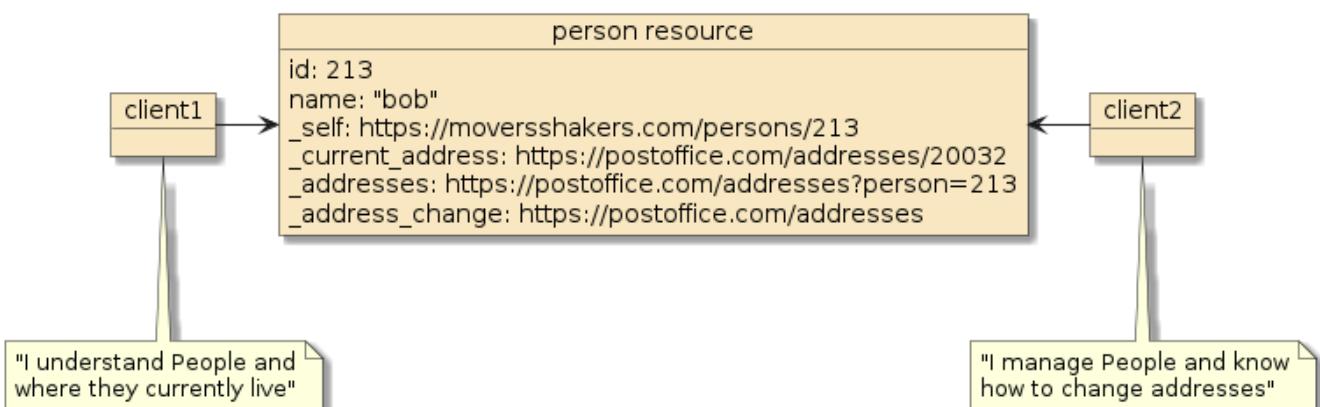


Figure 43. Example of Clients Dynamically Discovering State Represented through Hyperlinks

127.3. Static Interface Contracts

Dynamic discovery differs significantly from remote procedure call (RPC) techniques where static interface contracts are documented in detail to represent a certain level of capability offered by the server and understood by the client. A capability change rollout under the RPC approach may require coordination between all clients involved.

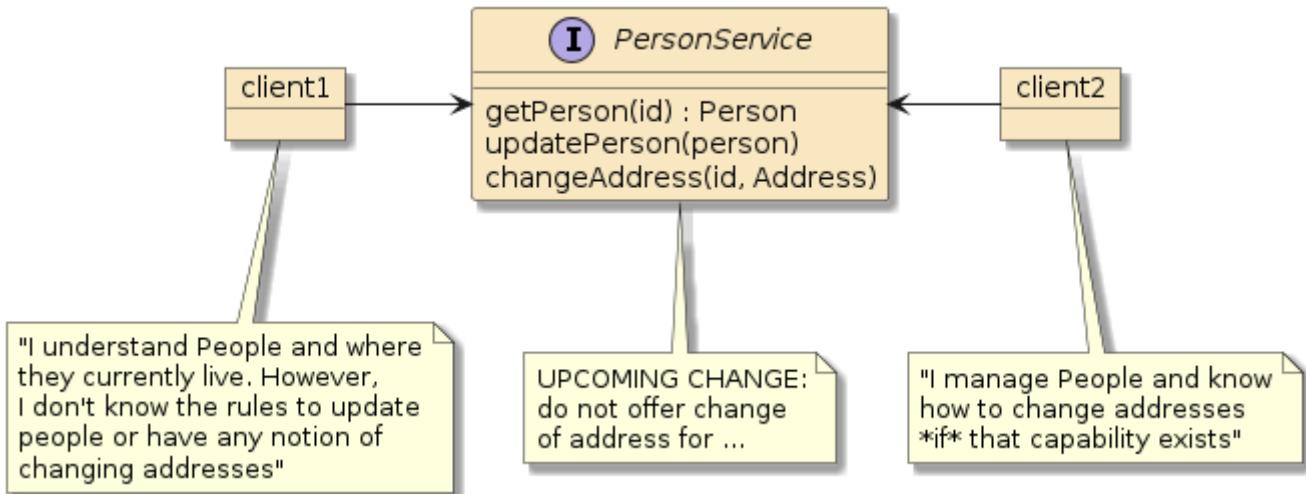


Figure 44. RPC Tight Coupling Example

127.4. Internet Scale

As clients morph from a few, well known sources to millions of lightweight apps running on end-user devices—the need to decouple service capability deployments through dynamic discovery becomes more important. Many features of REST provide this trait.



Do you have control of when clients update?

Design interfaces, clients, and servers with forward and backward compatibility in mind to allow for flexible rollout with minimal downtime.

127.5. How RESTful?

Many of the open and interfacing concepts of the WWW are attractive to today's service interface designers. However, implementing dynamic discovery is difficult—potentially making systems more complex and costly to develop. REST officially contains more than most interface designs use or possibly need to use. This causes developments to take only what they need—and triggers some common questions:

What is your definition of REST?

How RESTful are you?

127.6. Buzzword Association

For many developers and product advertisements eager to get their names associated with a modern and successful buzzword—REST to them is (incorrectly) anything using HTTP that is not SOAP. For others, their version of REST is (still incorrectly) anything that embraces much of the WWW but still lacks the rigor of making the interfaces dynamic through hyperlinks.

This places us in a state where most of the world refers to something as REST and RESTful when what they have is far from the official definition.

127.7. REST-like or HTTP-based

Giving a nod to this situation, we might use a few other terms:

- REST-like
- HTTP-based

Better yet and for more precise clarity of meaning, I like the definitions put forward in the Richardson Maturity Model (RMM).

127.8. Richardson MaturityModel (RMM)

The [Richardson Maturity Model \(RMM\)](#) was developed by Leonard Richardson and breaks down levels of RESTful maturity.^[4] Some of the old [CORBA](#) and [XML RPC](#) qualify for Level 0 only for the fact they adopt HTTP. However, they tunnel thru many WWW features in spite of using HTTP. Many modern APIs achieve some level of compliance with Levels 1 and 2, but rarely will achieve Level 3. However, that is okay because as you will see in the following sections—there are many worthwhile features in Level 2 without adding the complexity of HATEOAS.

Table 7. Richardson Maturity Model for REST

Level 3	<ul style="list-style-type: none">• using Hypermedia Controls i.e., the basis for Roy Fielding's definition of REST• dynamic discovery of state and capabilities thru hyperlinks
Level 2	<ul style="list-style-type: none">• using HTTP Methods i.e., handle similar situations in the same way• standardized methods and status codes• publicly expose method performed and status responses to better enable communication infrastructure support
Level 1	<ul style="list-style-type: none">• using Resources i.e., divide and conquer• e.g., rather than a single aggregate for endpoint calls, make explicit reference to lower-level targets
Level 0	<ul style="list-style-type: none">• using HTTP solely as a transport• e.g., CORBA tunneled all calls through HTTP POST

127.9. "REST-like"/"HTTP-based" APIs

Common "REST-like" or "HTTP-based" APIs are normally on a trajectory to strive for RMM Level 2 and are based on a few main principals included within the definition of REST.

- HTTP Protocol
- Resources
- URIs
- Standard HTTP Method and Status Code Vocabulary
- Standard Content Types for Representations

127.10. Uncommon REST Features Adopted

Links are used somewhat. However, they are rarely used in an opaque manner, rarely used within payloads, and rarely used with dynamic discovery. Clients commonly know the resources they are communicating with ahead of time and build URIs to those resources based on exposed details of the API and IDs returned in earlier responses. That is technically not a RESTful way to do things.

[1] "[Representational state transfer](#)" — Wikipedia Page

[2] "[Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation](#)", Roy Thomas Fielding, University of California, Irvine, 2000

[3] "[HATEOUS Wikipedia Page](#)"

[4] "[Richardson Maturity Model](#)", Martin Fowler, 2010

Chapter 128. RMM Level 2 APIs

Although I will commonly hear projects state that they implement a "REST" interface (and sometimes use it as "HTTP without SOAP"), I have rarely found a project that strives for dynamic discovery of resource capabilities as depicted by Roy Fielding and categorized by RMM Level 3.

These APIs try to make the most of HTTP and the WWW, thus at least making the term "HTTP-based" appropriate and RMM-level 2 a more accurate description. Acknowledging that there is technically one definition of REST and very few attempting to (or needing to) achieve it—I will be targeting RMM Level 2 for the web service interfaces developed in this course and will generically refer to them as "APIs".

At this point lets cover some of the key points of a RMM Level 2 API that I will be covering as a part of the course.

Chapter 129. HTTP Protocol Embraced

Various communications protocols have been transport agnostic. If you are old enough to remember [SOAP](#), you will have seen references to it being mapped to protocols other than HTTP (e.g., SOAP over JMS) and its use of HTTP lacked any leverage of WWW HTTP capabilities.

For SOAP and many other RPC protocols operating over HTTP—communication was tunnelled through HTTP POST messages, bypassing investments made in the existing and robust WWW infrastructure. For example, many requests for the same status of the same resource tunnelled thru POST messages would need to be answered again-and-again by the service. To fully leverage HTTP client-side and server-side caches, an alternative approach of exposing the status as a GET of a resource would save the responding service a lot of unnecessary work and speed up client.

REST communication technically does not exist outside of the HTTP transport protocol. Everything is expressed within the context of HTTP, leveraging the investment into the world's largest information system.

Chapter 130. Resource

By the time APIs reach RMM Level 1 compliance, service domains have been broken down into key areas, known as resources. These are largely noun-based (e.g., Documents, People, Companies), lower-level properties, or relationships. However, they go on to include actions or a long-running activity to be able to initiate them, monitor their status, and possibly perform some type of control.

Nearly anything can be made into a resource. HTTP has a limited number of methods but can have an unlimited number of resources. Some examples could be:

- products
- categories
- customers
- todos

130.1. Nested Resources

Resources can be nested under parent or related resources.

- categories/{id}
- categories/{id}/products
- todos/{name}
- todos/{name}/items

Chapter 131. Uniform Resource Identifiers (URIs)

Resources are identified using [Uniform Resource Identifier \(URIs\)](#).

A URI is a compact sequence of characters that identifies an abstract or physical resource.^[1]

— URI: Generic Syntax RFC Abstract 2005

URIs have a generic syntax composed of several components and are specialized by individual schemes (e.g., http, mailto, urn). The precise generic URI and scheme-specific rules guarantee uniformity of addresses.

Example URIs

```
https://github.com/spring-projects/spring-boot/blob/master/LICENSE.txt#L6 ①  
mailto:joe@example.com?cc=bob@example.com&body=hello ②  
urn:isbn:0-395-36341-1 ③
```

① example URL URI

② example email URI^[2]

③ example URN URI; "isbn" is a URN namespace^[1]

131.1. Related URI Terms

There are a few terms commonly associated with URI.

Uniform Resource Locator (URL)

URLs are a subset of URIs that provide a means to locate a specific resource by specifying primary address mechanism (e.g., network location).^[1]

Uniform Resource Name (URN)

URNs are used to identify resources without location information. They are a particular URI scheme. One common use of a URN is to define an XML namespace. e.g., `<core xmlns="urn:activemq:core">`.

URI reference

legal way to specify a full or relative URI

Base URI

leading components of the URI that form a base for additional layers of the tree to be appended

131.2. URI Generic Syntax

URI components are listed in hierarchical significance — from left to right — allowing for scheme-independent references to be made between resources in the hierarchy. The generic URI syntax and components are as follows: ^[3]

Generic URI components ^[3]

```
URI = scheme:[//authority]path[?query][#fragment]
```

The authority component breaks down into subcomponents as follows:

Authority Subcomponents ^[3]

```
authority = [userinfo@]host[:port]
```

Table 8. Generic URI Components

Scheme	sequence of characters, beginning with a letter followed by letters, digits, plus (+), period, or hyphen(-)
Authority	naming authority responsible for the remainder of the URI
User	how to gain access to the resource (e.g., username) - rare, authentication use deprecated
Host	case-insensitive DNS name or IP address
Port	port number to access authority/host
Path	identifies a resource within the scope of a naming authority. Terminated by the first question mark ("?"), pound sign ("#"), or end of URI. When the authority is present, the path must begin with a slash ("/") character
Query	indicated with first question mark ("?") and ends with pound sign ("#") or end of URI
Fragment	indicated with a pound("#") character and ends with end of URI

131.3. URI Component Examples

The following shows the earlier URI examples broken down into components.

Example URL URI Components

```
-- authority                                fragment --
/                                         \
https://github.com/spring-projects/spring-boot/blob/master/LICENSE.txt#L6
\                                         \
-- scheme          -- path
```

Path cannot begin with the two slash ("//") character string when the authority is not present.

Example mailto URI Components

```
-- path  
/  
mailto:joe@example.com?cc=bob@example.com&body=hello  
\  
-- scheme           -- query
```

Example URN URI Components

```
-- scheme  
/  
urn:isbn:0-395-36341-1  
\  
-- path
```

131.4. URI Characters and Delimiters

URI characters are encoded using [UTF-8](#). Component delimiters are slash (""/"), question mark ("?"'), and pound sign ("#"). Many of the other special characters are reserved for use in delimiting the sub-components.

Reserved Generic URI Delimiter Characters

```
: / @ [ ] ? ①
```

① square brackets("[]") are used to surround newer (e.g., IPv6) network addresses

Reserved Sub-delimiter Characters

```
! $ & ' ( ) * + , ; =
```

Unreserved URI Characters

```
alpha(A-Z,a-z), digit (0-9), dash(-), period(.), underscore(_), tilde(~)
```

131.5. URI Percent Encoding

(Case-insensitive) Percent encoding is used to represent characters reserved for delimiters or other purposes (e.g., %x2f and %xF both represent slash ("") character). Unreserved characters should not be encoded.

Example Percent Encoding

```
https://www.google.com/search?q=this%2Fthat ①
```

① slash("/") character is Percent Encoded as %2F

131.6. URI Case Sensitivity

Generic components like scheme and authority are case-insensitive but normalize to lowercase. Other components of the URI are assumed to be case-sensitive.

Example Case Sensitivity

```
HTTPS://GITHUB.COM/SPRING-PROJECTS/SPRING-BOOT ①  
https://github.com/SPRING-PROJECTS/SPRING-BOOT ②
```

① value pasted into browser

② value normalized by browser

131.7. URI Reference

Many times we need to reference a target URI and do so without specifying the complete URI. A URI reference can be the full target URI or a relative reference. A relative reference allows for a set of resources to reference one another without specifying a scheme or upper parts of the path. This also allows entire resource trees to be relocated without having to change relative references between them.

131.8. URI Reference Terms

target uri

the URI being referenced

Example Target URI

```
https://github.com/spring-projects/spring-boot/blob/master/LICENSE.txt#L6
```

network-path reference

relative reference starting with two slashes ("//"). My guess is that this would be useful in expressing a URI to forward to without wishing to express http versus https (i.e., "use the same scheme used to contact me")

Example Network Path Reference

```
//github.com/spring-projects/spring-boot/blob/master/LICENSE.txt#L6
```

absolute-path reference

relative reference that starts with a slash ("/"). This will be a portion of the URI that our API layer will be well aware of.

Example Absolute Path Reference

```
/spring-projects/spring-boot/blob/master/LICENSE.txt#L6
```

relative-path reference

relative reference that does not start with a slash ("/"). First segment cannot have a ":"—avoid confusion with scheme by prepending a "./" to the path. This allows us to express the branch of a tree from a point in the path.

Example Relative Path Reference

```
spring-boot/blob/master/LICENSE.txt#L6  
LICENSE.txt#L6  
./master/LICENSE.txt#L6
```

same-document reference

relative reference that starts with a pound ("#") character, supplying a fragment identifier hosted in the current URI.

Example Same Document Reference

```
#L6
```

base URI

leading components of the URI that form a base for additional layers of the tree to be appended

Example Base URI

```
https://github.com/spring-projects  
/spring-projects
```

131.9. URI Naming Conventions

Although URI specifications do not list path naming conventions and REST promotes opaque URIs—it is a common practice to name resource collections with a URI path that ends in a plural noun. The following are a few example absolute URI path references.

Example Resource Collection URI Absolute Path References

```
/api/products ①  
/api/categories  
/api/customers  
/api/todo_lists
```

① URI paths for resource collections end with a plural noun

Individual resource URIs are identified by an external identifier below the parent resource

collection.

Example Individual Resource Absolute URI Paths

```
/api/products/{productId} ①  
/api/categories/{categoryId}  
/api/customers/{customerId}  
/api/customers/{customerId}/sales
```

① URI paths for individual resources are scoped below parent resource collection URI

Nested resource URIs are commonly expressed as resources below their individual parent.

Example Nested Resource Absolute URI Paths

```
/api/products/{productId}/instructions ①  
/api/categories/{categoryId}/products  
/api/customers/{customerId}/purchases  
/api/todo_lists/{listName}/todo_items
```

① URI paths for resources of parent are commonly nested below parent URI

131.10. URI Variables

The query at the end of the URI path can be used to express optional and mandatory arguments. This is commonly used in queries.

Query Parameter Example

```
http://127.0.0.1:8080/jaxrsInventoryWAR/api/categories?name=&offset=0&limit=0  
name => (null) #has value null  
offset => 0  
limit => 0
```

Nested path parameters may express mandatory arguments.

Path Parameter Example

```
http://127.0.0.1:8080/jaxrsInventoryWAR/api/products/{id}  
http://127.0.0.1:8080/jaxrsInventoryWAR/api/products/1  
id => 1
```

[1] "Uniform Resource Identifier (URI): Generic Syntax RFC", Network Working Group, Berners-Lee, Fielding, Masinter, 2005

[2] "The 'mailto' URI Scheme", Duerst, Masinter, Zawinski, 2010

[3] [URI Wikipedia Page](#)

Chapter 132. Methods

HTTP contains a bounded set of methods that represent the "verbs" of what we are communicating relative to the resource. The bounded set provides a uniform interface across all resources.

There are four primary methods that you will see in most tutorials, examples, and application code.

Table 9. Primary HTTP Methods

GET	obtain a representation of resource using a non-destructive read
POST	create a new resource or tunnel a command to an existing resource
PUT	create a new resource with having a well-known identity or replace existing
DELETE	delete target resource

Example: Get Product ID=1

```
GET http://127.0.0.1:8080/jaxrsInventoryWAR/api/products/1
```

132.1. Additional HTTP Methods

There are two additional methods useful for certain edge conditions implemented by application code.

Table 10. Additional HTTP Methods

HEAD	logically equivalent to a GET without response payload - metadata only. Can provide efficient way to determine if resource exists and potentially last updated
PATCH	partial replace. Similar to PUT, but indicates payload provided does not represent the entire resource and may be represented as instructions of modifications to make. Useful hint for intermediate caches  The following post provides a good starting point to understanding the role of PATCH and the role of the JSON Patch and JSON Merge Patch instructions.

There are three more obscure methods used for debug and communication purposes.

Table 11. Communication Support Methods

OPTIONS	generates a list of methods supported for resource
TRACE	echo received request back to caller to check for changes
CONNECT	used to establish an HTTP tunnel — to proxy communications

Chapter 133. Method Safety

Proper execution of the internet protocols relies on proper outcomes for each method. With the potential of client-side proxies and server-side reverse proxies in the communications chain — one needs to pay attention to what can and should not change the state of a resource. "Method Safety" is a characteristic used to describe whether a method executed against a specific resource modifies that resource or has visible side effects.

133.1. Safe and Unsafe Methods

The following methods are considered "Safe" — thus calling them should not modify a resource and will not invalidate any intermediate cache.

- GET
- HEAD
- OPTIONS
- TRACE

The following methods are considered "Unsafe" — thus calling them is assumed to modify the resource and will invalidate any intermediate cache.

- POST
- PUT
- PATCH
- DELETE
- CONNECT

133.2. Violating Method Safety

Do not violate default method safety expectations

Internet communications is based upon assigned method safety expectations. However, these are just definitions. Your application code has the power to implement resource methods any way you wish and to knowingly or unknowingly violate these expectations. Learn the expected characteristics of each method and abide by them or risk having your API not immediately understood and render built-in Internet capabilities (e.g., caches) useless. The following are examples of what **not** to do:



Example Method Safety Violations

```
GET /jaxrsInventoryWAR/api/products/1?command=DELETE ①  
POST /jaxrsInventoryWAR/api/products/1 ②  
  content: {command: 'getProduct'}
```

- ① method violating GET Safety rule
- ② unsafe POST method tunneling safe GET command

Chapter 134. Idempotent

[Idempotence](#) describes a characteristic where a repeated event produces the same outcome every time executed. This is a very important concept in distributed systems that commonly have to implement eventual consistency—where failure recovery can cause unacknowledged commands to be executed multiple times.

The idempotent characteristic is independent of method safety. Idempotence only requires that the same result state be achieved each time called.

134.1. Idempotent and non-Idempotent Methods

The application code implementing the following HTTP methods should strive to be idempotent.

- GET
- PUT
- DELETE
- HEAD
- OPTIONS

The following HTTP methods are defined to not be idempotent.

- POST
- PATCH
- CONNECT

Relationship between Idempotent and browser page refresh warnings?

The standard convention of Internet protocol is that most methods except for POST are assumed to be idempotent. That means a page refresh for a page obtained from a GET gets immediately refreshed and a warning dialogue is displayed if it was the result of a POST.



Chapter 135. Response Status Codes

Each HTTP response is accompanied by a standard [HTTP status code](#). This is a value that tells the caller whether the request succeeded or failed and a type of success or failure.

Status codes are separated into five (5) categories

- 1xx - informational responses
- 2xx - successful responses
- 3xx - redirected responses
- 4xx - client errors
- 5xx - server errors

135.1. Common Response Status Codes

The following are common response status codes

Table 12. Common HTTP Response Status Codes

Code	Name	Meaning
200	OK	"We achieved what you wanted - may have previously done this"
201	CREATED	"We did what you asked and a new resource was created"
202	ACCEPTED	"We received your request and will begin processing it later"
204	NO_CONTENT	"Just like a 200 with an empty payload, except the status makes this clear"
400	BAD_REQUEST	"I do not understand what you said and never will"
401	UNAUTHORIZED	"We need to know who you are before we do this"
403	FORBIDDEN	"We know who you are and you cannot say what you just said"
404	NOT_FOUND	"We could not locate the target resource of your request"
422	UNPROCESSABLE_ENTITY	"I understood what you said, but you said something wrong"
500	INTERNAL_ERROR	"Ouch! Nothing wrong with what you asked for or supplied, but we currently have issues completing. Try again later and we may have this fixed."

Chapter 136. Representations

Resources may have multiple independent representations. There is no direct tie between the data format received from clients, returned to clients, or managed internally. Representations are exchanged using standard [MIME or Media types](#). Common media types for information include

- application/json
- application/xml
- text/plain

Common data types for raw images include

- image/jpg
- image/png

136.1. Content Type Headers

Clients and servers specify the type of content requested or supplied in header fields.

Table 13. HTTP Content Negotiation Headers

Accept	defines a list of media types the client understands, in priority order
Content-Type	identifies the format for data supplied in the payload

In the following example, the client supplies a representation in `text/plain` and requests a response in XML or JSON—in that priority order. The client uses the Accept header to express which media types it can handle and both use the Content-Type to identify the media type of what was provided.

Example Accept and Content-Type Headers

```
> POST /greeting/hello
> Accept: application/xml,application/json
> Content-Type: text/plain
hi

< 200/OK
< Content-Type: application/xml
<greeting type="hello" value="hi"/>
```

The next exchange is similar to the previous example, with the exception that the client provides no payload and requests JSON or anything else (in that priority order) using the Accept header. The server returns a JSON response and identifies the media type using the Content-Type header.

Example JSON Accept and Content-Type Headers

```
> GET /greeting/hello?name=jim
```

```
> Accept: application/json,*/*
```

```
< 200/OK
```

```
< Content-Type: application/json
```

```
{ "msg" : "hi, jim" }
```

Chapter 137. Links

RESTful applications dynamically express their state through the use of hyperlinks. That is an RMM Level 3 characteristic use of links. As mentioned earlier, REST-like APIs do not include that level of complexity. If they do use links, these links will likely be constrained to standard response headers.

The following is an example partial POST response with links expressed in the header.

Example Response Headers with Links

```
POST http://localhost:8080/ejavaTodos/api/todo_lists
{"name":"My First List"}
=> Created/201
Location: http://localhost:8080/ejavaTodos/api/todo_lists/My%20First%20List ①
Content-Location: http://localhost:8080/ejavaTodos/api/todo_lists/My%20First%20List ②
```

① Location expresses the URI to the resource just acted upon

② Content-Location expresses the URI of the resource represented in the payload

Chapter 138. Summary

In this module, we learned that:

- technically — terms "REST" and "RESTful" have a specific meaning defined by Roy Fielding
- the Richardson Maturity Model (RMM) defines several levels of compliance to RESTful concepts, with level 3 being RESTful
 - very few APIs achieve full RMM level 3 RESTful adoption
 - but that is OK!!! — there are many useful and powerful WWW constructs easily made available before reaching the RMM level 3
 - can be referred to as "REST-like", "HTTP-based", or "RMM level 2"
 - marketers of the world attempting to leverage a buzzword, will still call them REST APIs
- most serious REST-like APIs adopt
 - HTTP
 - multiple resources identified through URIs
 - HTTP-compliant use of methods and status codes
 - method implementations that abide by defined safety and idempotent characteristics
 - standard resource representation formats like JSON, XML, etc.

Spring MVC

copyright © 2024 jim stafford (jim.stafford@jhu.edu)

Chapter 139. Introduction

You learned the meaning of web APIs and supporting concepts in the previous lecture. This module is an introductory lesson to get started implementing some of those concepts. Since this lecture is primarily implementation, I will use a set of simplistic remote procedure calls (RPC) that are **far** from REST-like and place the focus on making and mapping to HTTP calls from clients to services using Spring and Spring Boot.

139.1. Goals

The student will learn to:

- identify two primary paradigms in today's server logic: synchronous and reactive
- develop a service accessed via HTTP
- develop a client to an HTTP-based service
- access HTTP response details returned to the client
- explicitly supply HTTP response details in the service code

139.2. Objectives

At the conclusion of this lecture and related exercises, the student will be able to:

1. identify the difference between the Spring MVC and Spring WebFlux frameworks
2. identify the difference between synchronous and reactive approaches
3. identify reasons to choose synchronous or reactive
4. implement a service method with Spring MVC synchronous annotated controller
5. implement a synchronous client using RestTemplate API
6. implement a synchronous client using RestClient fluent API
7. implement a client using Spring Webflux fluent API in synchronous mode
8. pass parameters between client and service over HTTP
9. return HTTP response details from service
10. access HTTP response details in client
11. implement exception handler outside of service method

Chapter 140. Spring Web APIs

There are two primary, overlapping frameworks within Spring for developing HTTP-based APIs:

- [Spring MVC](#)
- [Spring WebFlux](#)

Spring MVC is the legacy framework that operates using synchronous, blocking request/reply constructs. Spring WebFlux is the follow-on framework that builds on Spring MVC by adding asynchronous, non-blocking constructs that are inline with the [reactive streams paradigm](#).

140.1. Lecture/Course Focus

The focus of this lecture, module, and most portions of the course will be on synchronous communications patterns. The synchronous paradigm is simpler, and there are a ton of API concepts to cover before worrying about managing the asynchronous streams of the reactive programming model. In addition to reactive concepts, Spring WebFlux brings in a heavy dose of Java 8 lambdas and functional programming that should only be applied once we master more of the API concepts.

However, we need to know the two approaches exist to make sense of the software and available documentation. For example, the long-time legacy client-side of Spring MVC (i.e., [RestTemplate](#)) was put in "maintenance mode" (minor changes and bug fixes only) towards the end of Spring 5, with its duties fulfilled by Spring WebFlux (i.e., [WebClient](#)). Spring 6 introduced a middle ground with [RestClient](#) that addresses the synchronous communication simplicity of [RestTemplate](#) with the fluent API concepts of [WebClient](#).

It is certain that you will encounter use of [RestTemplate](#) in legacy Spring applications and there is no strong reason to replace. There is a good chance you may have the desire to work with a fluent or reactive API. Therefore, I will be demonstrating synchronous client concepts using each library to help cover all bases.



WebClient examples demonstrated here are intentionally synchronous

Examples of Spring WebFlux's [WebClient](#) will be demonstrated as a synchronous replacement for Spring MVC [RestTemplate](#). Details of the reactive API will not be covered.

140.2. Spring MVC

[Spring MVC](#) was originally implemented for writing Servlet-based applications. The term "MVC" stands for "Model, View, and Controller"—which is a standard framework pattern that separates concerns between:

- data and access to data ("the model"),
- representation of the data ("the view"), and
- decisions of what actions to perform when ("the controller").

The separation of concern provides a means to logically divide web application code along architecture boundaries. Built-in support for HTTP-based APIs has matured over time, and with the shift of UI web applications to JavaScript frameworks running in the browser, the focus has likely shifted towards the API development.

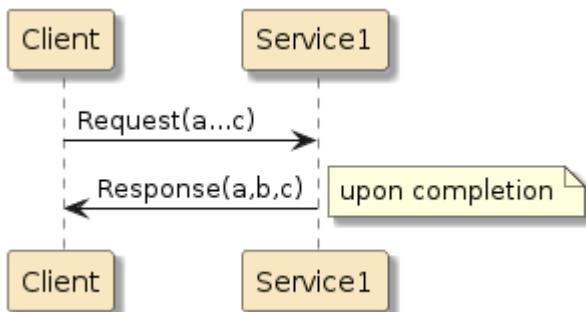


Figure 45. Spring MVC Synchronous Model

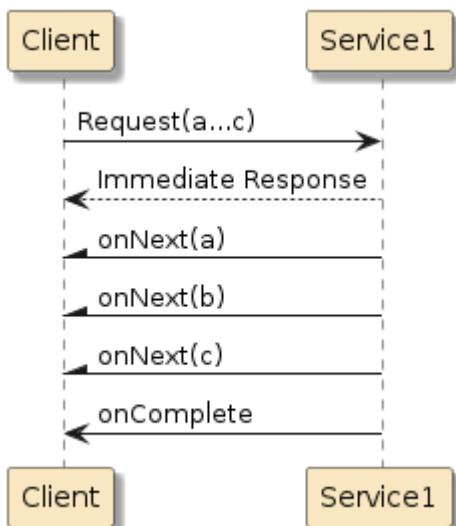
As mentioned earlier, the programming model for Spring MVC is synchronous, blocking request/reply. Each active request is blocked in its own thread while waiting for the result of the current request to complete. This mode scales primarily by adding more threads—most of which are blocked performing some sort of I/O operation.

140.3. Spring WebFlux

[Spring WebFlux](#) is built using a stream-based, reactive design as a part of Spring 5/Spring Boot 2. The [reactive programming model](#) was adopted into the [java.util.concurrent package](#) in Java 9, to go along with other asynchronous programming constructs—like [Future<T>](#).

Some of the core concepts—like annotated [@RestController](#) and method associated annotations—still exist. The most visible changes added include the optional functional controller and the new, mandatory data input and return publisher types:

- [Mono](#) - a handle to a promise of a single object in the future
- [Flux](#) - a handle to a promise of many objects in the future



For any single call, there is an immediate response and then a flow of events that start once the flow is activated by a subscriber. The flow of events is published to and consumed from the new mandatory Mono and Flux data input and return types. No overall request is completed using an end-to-end single thread. Work to process each event must occur in a non-blocking manner. This technique sacrifices raw throughput of a single request to achieve better performance when operating at a greater concurrent scale.

Figure 46. Spring WebFlux Reactive Model

140.4. Synchronous vs. Asynchronous

To go a little further in contrasting the two approaches, the diagram below depicts a contrast

between a call to two separate services using the synchronous versus asynchronous processing paradigms.

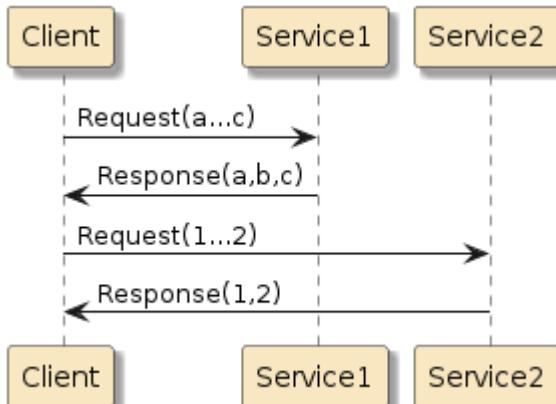


Figure 47. Synchronous

For synchronous, the call to service 2 cannot be initiated until the synchronous call/response from service 1 is completed

For asynchronous, the calls to service 1 and 2 are initiated sequentially but are carried out concurrently, and completed independently

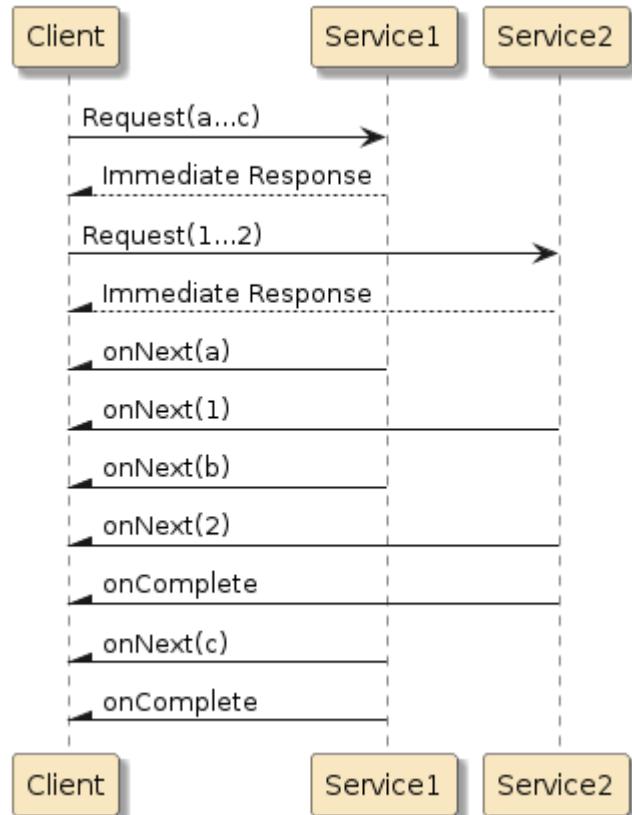


Figure 48. Asynchronous

There are different types of asynchronous processing. Spring has long supported threads with `@Async` methods. However, that style simply launches one or more additional threads that potentially also contain synchronous logic that will likely block at some point. The reactive model is strictly non-blocking—relying on the backpressure of available data and the resources being available to consume it. With the reactive programming paradigm comes strict rules of the road.

140.5. Mixing Approaches

There is a certain amount of mixture of approaches allowed with Spring MVC and Spring WebFlux. A pure reactive design without a trace of Spring MVC can operate on the `Reactor Netty` engine—optimized for reactive processing. Any use of Web MVC will cause the application to be considered a Web MVC application, choose between Tomcat or Jetty for the web server, and operate any use of reactive endpoints in a compatibility mode.^[1]

With that said—functionally, we can mix Spring Web MVC and Spring WebFlux together in an application using what is considered to be the Web MVC container.

- Synchronous and reactive flows can operate side-by-side as independent paths through the code
- Synchronous flows can make use of asynchronous flows. A primary example of that is using the `WebClient` reactive methods from a Spring MVC controller-initiated flow

However, we cannot have the callback of a reactive flow make synchronous requests that can indeterminately block—or it itself will become synchronous and tie up a critical reactor thread.

Spring MVC has non-optimized, reactive compatibility



Tomcat and Jetty are Spring MVC servlet engines. Reactor Netty is a Spring WebFlux engine. Use of reactive streams within the Spring MVC container is supported—but not optimized or recommended beyond use of the [WebClient](#) in Spring MVC applications. Use of synchronous flows is not supported by Spring WebFlux.

140.6. Choosing Approaches

Independent synchronous and reactive flows can be formed on a case-by-case basis and optimized if implemented on separate instances. ^[1] We can choose our ultimate solution(s) based on some of the recommendations below.

Synchronous

- existing synchronous API working fine—no need to change ^[2]
- easier to learn - can use standard Java imperative programming constructs
- easier to debug - everything in the same flow is commonly in the same thread
- the number of concurrent users is a manageable (e.g., <100) number ^[3]
- service is CPU-intensive ^[4]
- codebase makes use of ThreadLocal
- service makes use of synchronous data sources (e.g., JDBC, JPA)

Reactive

- need to serve a significant number (e.g., 100-300) of concurrent users ^[3]
- requires knowledge of Java stream and functional programming APIs
- does little to no good (i.e., **badly**) if the services called are synchronous (i.e., initial response returns when overall request complete) (e.g., JDBC, JPA)
- desire to work with Kotlin or Java 8 lambdas ^[2]
- service is IO-intensive (e.g., database or external service calls) ^[4]

For many of the above reasons, we will start out our HTTP-based API coverage in this course using the synchronous approach.

[1] ["Can I use SpringMvc and webflux together?", Brian Clozel, 2018](#)

[2] ["Spring WebFlux Documentation - Applicability", version 5.2.6 release](#)

[3] ["SpringBoot: Performance War", Santhosh Krishnan, 2020](#)

[4] ["Do's and Don'ts: Avoiding First-Time Reactive Programmer Mines", Sergei Egorov, SpringOne Platform, 2019](#)

Chapter 141. Maven Dependencies

Most dependencies for Spring MVC are satisfied by changing `spring-boot-starter` to `spring-boot-starter-web`. Among other things, this brings in dependencies on `spring-webmvc` and `spring-boot-starter-tomcat`.

Spring MVC Starter Dependency

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

The dependencies for Spring MVC and Spring WebFlux's `WebClient` are satisfied by adding `spring-boot-starter-webflux`. It primarily brings in the `spring-webflux` and the reactive libraries, and `spring-boot-starter-reactor-netty`. We won't be using the netty engine, but `WebClient` does make use of some netty client libraries that are brought in when using the starter.

Spring MVC/Spring WebFlux Blend Dependency

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
```

Chapter 142. Sample Application

To get started covering the basics of Web MVC, I am going to use a basic, remote procedure call (RPC)-oriented, [RMM level 1](#) example where the web client simply makes a call to the service to say "hi". The example is located within the `rpc-greeter-svc` module.

```
|-- pom.xml
`-- src
  |-- main
  |   |-- java
  |   |   '-- info
  |   |       '-- ejava
  |   |           '-- examples
  |   |               '-- svc
  |   |                   '-- rpc
  |   |                       |-- GreeterApplication.java
  |   |                   '-- greeter
  |   |                       '-- controllers ①
  |   |                           '-- RpcGreeterController.java
  |   '-- resources
  |       '-- ...
`-- test
  |-- java
  |   '-- info
  |       '-- ejava
  |           '-- examples
  |               '-- svc
  |                   '-- rpc
  |                       '-- greeter ②
  |                           |-- GreeterRestTemplateNTest.java
  |                           |-- GreeterRestClientNTest.java
  |                           |-- GreeterSyncWebClientNTest.java
  |
  |                           |-- GreeterHttpIfaceNTest.java
  |                           |-- GreeterAPI.java
  |
  |                   '-- ClientTestConfiguration.java
  '-- resources
      '-- ...
```

① example @RestController

② example clients using RestTemplate, RestClient, WebClient, and Http Interface Proxy

Chapter 143. Annotated Controllers

Traditional Spring MVC APIs are primarily implemented around annotated controller components. Spring has a hierarchy of annotations that help identify the role of the component class. In this case the controller class will commonly be annotated with `@RestController`, which wraps `@Controller`, which wraps `@Component`. This primarily means that the class will get automatically picked up during the component scan if it is in the application's scope.

Example Spring MVC Annotated Controller

```
package info.ejava.examples.svc.httpapi.greeter.controllers;

import org.springframework.web.bind.annotation.RestController;

@RestController
// ==> wraps @Controller
//      ==> wraps @Component
public class RpcGreeterController {
    //...
}
```

143.1. Class Mappings

Class-level mappings can be used to establish a base definition to be applied to all methods and extended by method-level annotation mappings. Knowing this, we can define the base URI path using a `@RequestMapping` annotation on the controller class and all methods of this class will either inherit or extend that URI path.

In this particular case, our class-level annotation is defining a base URL path of `/rpc/greeting`.

Example Class-level Mapping

```
...
import org.springframework.web.bind.annotation.RequestMapping;

@RestController
@RequestMapping("rpc/greeter") ①
public class RpcGreeterController {
    ...
}
```

① `@RequestMapping.path="rpc/greeting"` at class level establishes base URI path for all hosted methods

Annotations can have alias and defaults



- `value` is an alias for `path` in the `@RequestMapping` annotation
- any time there is a **single** value expressed without a property name within an annotation, the `omitted name defaults to value`

We can use either `path`, `value`, or no name (when nothing else supplied) to express the path in `@RequestMapping`.



Annotating class can help keep from repeating common definitions

Annotations like `@RequestMapping`, applied at the class level establish a base path for all HTTP-accessible methods of the class.

143.2. Method Request Mappings

There are two initial aspects to map to our method in our first simple example: URI and HTTP method.

Example Endpoint URI

```
GET /rpc/greeter/sayHi
```

- URI - we already defined a base URI path of `/rpc/greeter` at the class level—we now need to extend that to form the final URI of `/rpc/greeter/sayHi`
- HTTP method - this is specific to each class method—so we need to explicitly declare GET (one of the standard `RequestMethod` enums) on the class method

Example Endpoint Method Implementation

```
...
/** 
 * This is an example of a method as simple as it gets
 * @return hi
 */
@RequestMapping(path="sayHi", ①
    method=RequestMethod.GET) ②
public String sayHi() {
    return "hi";
}
```

① `@RequestMapping.path` at the method level appends `sayHi` to the base URI

② `@RequestMapping.method=GET` registers this method to accept HTTP GET calls to the URI `/rpc/greeter/sayHi`



`@GetMapping` is an alias for `@RequestMapping(method=GET)`

Spring MVC also defines a `@GetMapping` and other HTTP method-specific annotations that simply wraps `@RequestMapping` with a specific method value (e.g., `method=GET`). We can use either form at the method level.

143.3. Default Method Response Mappings

A few of the prominent response mappings can be determined automatically by the container in

simplistic cases:

response body

The response body is automatically set to the marshalled value returned by the endpoint method. In this case, it is a literal String mapping.

status code

The container will return the following default status codes

- 200/OK - if we return a non-null value
- 404/NOT_FOUND - if we return a null value
- 500/INTERNAL_SERVER_ERROR - if we throw an exception

Content-Type header

The container sensibly mapped our returned String to the `text/plain` Content-Type.

Example Response Mappings Result

```
< HTTP/1.1 200 ①
< Content-Type: text/plain;charset=UTF-8 ②
< Content-Length: 2
...
hi ③
```

① non-null, no exception return mapped to HTTP status 200

② non-null `java.lang.String` mapped to `text/plain` content type

③ value returned by endpoint method

143.4. Executing Sample Endpoint

Once we start our application and enter the following in the browser, we get the expected string "hi" returned.

Example Endpoint Output

```
http://localhost:8080/rpc/greeter/sayHi

hi
```

If you have access to `curl` or another HTTP test tool, you will likely see the following additional detail.

Example Endpoint HTTP Exchange

```
$ curl -v http://localhost:8080/rpc/greeter/sayHi
...
> GET /rpc/greeter/sayHi HTTP/1.1
> Host: localhost:8080
```

```
> User-Agent: curl/7.54.0
> Accept: /*
>
< HTTP/1.1 200
< Content-Type: text/plain; charset=UTF-8
< Content-Length: 2
...
hi
```

Chapter 144. RestTemplate Client

The primary point of making a callable HTTP endpoint is the ability to call that endpoint from another application. With a functional endpoint ready to go, we are ready to create a Java client and will do so within a JUnit test using Spring MVC's `RestTemplate` class in the simplest way possible.

Please note that most of these steps are true for any Java HTTP client we might use. I will go through all the steps for `RestTemplate` here but only cover the unique aspects to the alternate techniques later on.

144.1. JUnit Integration Test Setup

We start our example by creating an integration unit test. That means we will be using the Spring context and will do so using `@SpringBootTest` annotation with two key properties:

- classes - reference `@Component` and/or `@Configuration` class(es) to define which components will be in our Spring context (default is to look for `@SpringBootConfiguration`, which is wrapped by `@SpringBootApplication`).
- webEnvironment - to define this as a web-oriented test and whether to have a fixed (e.g., 8080), random, or none for a port number. The random port number will be injected using the `@LocalServerPort` annotation. The default value is MOCK—for Mock test client libraries able to bypass networking.

Example JUnit Integration Unit Test Setup

```
package info.ejava.examples.svc.rpc.greeter;

import info.ejava.examples.svc.rpc.GreeterApplication;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.web.server.LocalServerPort;

@SpringBootTest(classes = GreeterApplication.class, ①
               webEnvironment = SpringApplication.WebEnvironment.RANDOM_PORT) ②
public class GreeterRestTemplateNTest {
    @LocalServerPort ③
    private int port;
```

① using the application to define the components for the Spring context

② the application will be started with a random HTTP port#

③ the random server port# will be injected into `port` annotated with `@LocalServerPort`

`@LocalServerPort` is alias for Property `local.server.port`

`@LocalServerPort` annotation acts as an alias for the `local.server.port` property.



```
package org.springframework.boot.test.web.server;
import org.springframework.beans.factory.annotation.Value;
```

```
...
@Value("${local.server.port}")
public @interface LocalServerPort {}
```

One could use that property instead to express the injection.

```
@Value("${local.server.port}")
private int port;
```

LocalServerPort Injection Alternatives

`@LocalServerPort` is not available until the web components are fully initialized—which constrains how we can inject.

As you saw earlier, we can have it injected as an attribute of the test case class. This would be good if many of the `@Test` methods needed access to the raw port value.

Inject as Test Attribute

```
@SpringBootTest(...)
public class GreeterRestTemplateNTest {
    @LocalServerPort
    private int port; //inject option way1
```

A close alternative would be to inject the value into the `@BeforeEach` lifecycle method. This would be good if `@Test` methods did not use the raw port value—but may use something that was built from the value.



Inject into Test Lifecycle Methods

```
@BeforeEach
public void init(@LocalServerPort int port) { //inject option way2
    baseUrl = String.format("http://localhost:%d/rpc/greeter",
    port);
}
```

We could move the injection to the `@TestConfiguration`. However, since the configuration is read in before the test is initialized, we must inject it into `@Bean` factory methods (versus an attribute) and annotate the `@Bean` factory with `@Lazy`. Lazy bean factories are called on demand versus eagerly at startup.

Create `@Bean` Factory using `@LocalServerPort` and `@Lazy`

```
import org.springframework.context.annotation.Lazy;
...
@TestConfiguration(proxyBeanMethods = false)
public class ClientTestConfiguration {
```

```

@Lazy
public String baseUrl(@LocalServerPort int port) {//inject option
way3
    return String.format("http://localhost:%d/rpc/greeter", port);
}

```

Inject @Bean into Test Case

```

@SpringBootTest(classes = {GreeterApplication.class, //optionally
naming app config
ClientTestConfiguration.class},
webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class GreeterRestTemplateNTest {
    @Autowired @Qualifier("baseUrl") //qualifier makes bean selection
more explicit
    private String injectedBaseUrl; //initialized in test config using
way3

```

144.2. Form Endpoint URL

Next, we will form the full URL for the target endpoint. We can take the parts we know and merge that with the injected server port number to get a full URL.

Forming endpoint URL with String.format()

```

@LocalServerPort
private int port;

@Test
public void say_hi() {
    //given - a service available at a URL and client access
    String url = String.format("http://localhost:%d/rpc/greeter/sayHi", port); ①
    ...
}

```

① full URL to the example endpoint

Starting Simple

Starting simple. We will be covering more type-safe, purpose-driven ways to perform related client actions in this and follow-on lectures.



144.3. Obtain RestTemplate

With a URL in hand, we are ready to make the call. We will do that first using the synchronous `RestTemplate` from the Spring MVC library.

Spring's `RestTemplate` is a thread safe class that can be constructed with a default constructor for the simple case—or through a `builder` in more complex cases and injected to take advantage of

separation of concerns.

Example Obtain Simple/Default RestTemplate

```
import org.springframework.web.client.RestTemplate;  
...  
RestTemplate restTemplate = new RestTemplate();
```

144.4. Invoke HTTP Call

There are dozens of potential calls we can make with `RestTemplate`. We will learn many more, but in this case we are

- performing an HTTP GET
- executing the HTTP method against a URL
- returning the response body content as a String

Example Invoke HTTP Call

```
String greeting = restTemplate  
    .getForObject(url, String.class); ①
```

① return a String greeting from the response body of a GET URL call

144.4.1. Exceptions

Note that a successful return from `getForObject()` will only occur if the response from the server is a 2xx/successful response. Otherwise, an exception of one of the following types will be thrown:

- `RestClientException` - error occurred communicating with server
 - `RestClientResponseException` error response received from server
 - `HttpStatusErrorException` - HTTP response received and HTTP status known
 - `HttpServerErrorException` - HTTP server (5xx) errors
 - `HttpClientErrorException` - HTTP client (4xx) errors
 - BadRequest, NotFound, UnprocessableEntity, ...

144.5. Evaluate Response

At this point, we have made our request and have received our reply and can evaluate the reply against what was expected.

Evaluate Response Body

```
//then - we get a greeting response  
then(greeting).isEqualTo("hi");
```

Chapter 145. Spring Rest Clients

The Spring 5 documentation stated [RestTemplate](#) was going into "maintenance mode" and that we should switchover to using the Spring WebFlux [WebClient](#). The current [Spring 6 documentation](#) dropped that guidance and made the choice driven by:

- synchronous - [RestTemplate](#)
- fluent and synchronous - [RestClient](#), new in [Spring 6.1](#)
- fluent and asynchronous/reactive - [WebClient](#)

Spring 6 also added features to all three for:

- client-side API facade - [HTTP Interface](#) - provides a type-safe business interface to any of the clients

I will summarize these additions next.

Chapter 146. RestClient Client

`RestClient` is a synchronous API like `RestTemplate`, but works using fluent ("chaining"; `client.m1().m2()`) API calls like `WebClient`. The asynchronous `WebClient` fluent API was introduced in Spring 5 and `RestClient` followed in Spring 6.1. When using `WebClient` in synchronous mode—the primary difference with `RestClient` is no need to explicitly block for exchanges to complete.

In demonstrating `RestClient`, there are a few aspects of our `RestTemplate` example that do not change and I do not need to repeat.

- JUnit test setup—i.e., establishing the Spring context and random port#
- Obtaining a URL
- Evaluating returned response

The new aspects include

- obtaining the `RestClient` instance
- invoking the HTTP endpoint and obtaining result

146.1. Obtain RestClient

`RestClient` is an interface and must be constructed through a builder. A default builder can be obtained through a static method of the `RestClient` interface. `RestClient` is also thread safe, is capable of being configured in a number of ways, and its builder can be injected to create individualized instances.

Example Obtain RestClient

```
import org.springframework.web.client.RestClient;
...
RestClient restClient = RestClient.builder().build();
```

If you are already invested in a detailed `RestTemplate` setup of configured defaults and want the fluent API, `RestClient` can be constructed from an existing `RestTemplate` instance.

Example Building RestClient from RestTemplate

```
RestTemplate restTemplate = ...
RestClient restClient=RestClient.create(restTemplate);
```

146.2. Invoke HTTP Call

The methods for `RestClient` are arranged in a builder type pattern where each layer of call returns a type with a constrained set of methods that are appropriate for where we are in the call tree.

The example below shows an example of:

- performing an HTTP GET
- targeting the HTTP methods at a specific URL
- retrieving an overall result—which is really a demarcation that the request definition is complete and from here on is the definition for what to do with the response
- retrieving the body of the result—a specification of the type to expect

Example Invoke HTTP Call

```
String greeting = restClient.get()  
    .uri(url)  
    .retrieve()  
    .body(String.class);
```

Chapter 147. WebClient Client

`WebClient` and `RestClient` look and act very much the same, with the primary difference being the reactive/asynchronous API aspects for `WebClient`.

147.1. Obtain WebClient

`WebClient` is an interface and must be constructed through a builder. A default builder can be obtained through a static method of the `WebClient` interface. `WebClient` is also thread safe, is capable of being configured in a number of ways, and its builder can be injected to create individualized instances.

Example Obtain WebClient

```
import org.springframework.web.reactive.function.client.WebClient;  
...  
WebClient webClient = WebClient.builder().build();
```

One cannot use a `RestTemplate` or `RestClient` instance to create a `WebClient`. They are totally different threading models under the hood.

147.2. Invoke HTTP Call

The fluent API methods for `WebClient` are much the same as `RestClient` except for when it comes to obtaining the payload body.

The example below shows an example of:

- performing an HTTP GET
- targeting the HTTP methods at a specific URL
- retrieving an overall result—which is really a demarcation that the request definition is complete and from here on is the definition for what to do with the response
- retrieving the body of the result—a specification of what to do with the response when it arrives. This will be a publisher (e.g., `Mono` or `Flux`) of some sort of value or type based on the response
- blocking until the reactive response is available

Example Invoke HTTP Call

```
String greeting = webClient.get()  
    //same  
    .uri(url)  
    //same  
    .retrieve()  
    //same  
    .bodyToMono(String.class)  
    .block(); ①
```

① Calling `block()` causes the reactive flow definition to begin producing data

The `block()` call is the synchronous part that we would look to avoid in a truly reactive thread. It is a type of subscriber that triggers the defined flow to begin producing data. This `block()` is blocking the current (synchronous) thread—just like `RestTemplate`. The portions of the call ahead of `block()`

are performed in a reactive set of threads.

Chapter 148. Spring HTTP Interface

This last feature ([HTTP Interface](#)) allows you to define a typed interface for your client API using a Java interface, annotations, and any of the Spring client APIs we have just discussed. Spring will implement the details using dynamic proxies (discussed in detail much later in the course).

We can define a simple example using our `/sayHi` endpoint by defining a method with the information required to make the HTTP call. This is very similar to what is defined on the server-side.

Example Type-safe Client Interface

```
import org.springframework.web.service.annotation.GetExchange;

interface MyGreeter {
    @GetExchange("/sayHi")
    String sayHi();
}
```

We then build a `RestTemplate`, `RestClient`, or `WebClient` by any means and assign it a `baseUrl`. The `baseUrl` plus `@GetExchange` value must equal the server-side URL.

Build Client with Base URL

```
String url = ...
RestClient restClient = RestClient.builder().baseUrl(url).build();
```

We then can create an instance of the interface using the lower-level API, `RestClientAdapter`, and `HttpServiceProxyFactory`.

Create Proxy

```
import org.springframework.web.client.support.RestClientAdapter;
import org.springframework.web.service.invoker.HttpServiceProxyFactory;
...
RestClientAdapter adapter = RestClientAdapter.create(restClient);
HttpServiceProxyFactory factory = HttpServiceProxyFactory.builderFor(adapter).build();
MyGreeter greeterAPI = factory.createClient(MyGreeter.class);
```

At this point we can call it like any Java instance/method.

Example Call to Spring HTTP Interface

```
//when - calling the service
String greeting = greeterAPI.sayHi();
```

The Spring HTTP Interface is extremely RPC-oriented, but we can make it REST-like enough to be useful. Later examples in this lecture will show some extensions.

Chapter 149. Implementing Parameters

There are three primary ways to map an HTTP call to method input parameters:

- request body—annotated with `@RequestBody` that we will see in a POST
- path parameter—annotated with `@PathVariable`
- query parameter - annotated with `@RequestParam`

The later two are part of the next example and expressed in the URI.

Example URI with path and query parameters

```
/ ①  
GET /rpc/greeter/say/hello?name=jim  
      \ ②
```

- ① URI path segments can be mapped to input method parameters
- ② individual query values can be mapped to input method parameters

- we can have 0 to N path or query parameters
 - path parameters are part of the resource URI path and are commonly required when defined—but that is not a firm rule
 - query parameters are commonly the technique for optional arguments against the resource expressed in the URI path

149.1. Controller Parameter Handling

Parameters derived from the URI path require that the path be expressed with `{placeholder}` names within the string. That placeholder name will be mapped to a specific method input parameter using the `@PathVariable` annotation. In the following example, we are mapping whatever is in the position held by the `{greeting}` placeholder—to the `greeting` input variable.

Specific query parameters are mapped by their name in the URL to a specific method input parameter using the `@RequestParam` annotation. In the following example, we are mapping whatever is in the value position of `name=` to the `name` input variable.

Example Path and Query Param

```
@RequestMapping(path="say/{greeting}", ①  
    method=RequestMethod.GET)  
public String sayGreeting(  
    @PathVariable("greeting") String greeting, ①  
    @RequestParam(value = "name", defaultValue = "you") String name) { ②  
    return greeting + ", " + name;  
}
```

- ① URI path placeholder `{greeting}` is being mapped to method input parameter `String greeting`
- ② URI query parameter `name` is being mapped to method input parameter `String name`

No direct relationship between placeholder/query names and method input parameter names



There is no direct correlation between the path placeholder or query parameter name and the name of the variable without the `@PathVariable` and `@RequestParam` mappings. Having them match makes the mental mapping easier, but the value for the internet client URI name may not be the best value for the internal Java controller variable name.

149.2. Client-side Parameter Handling

As mentioned above, the path and query parameters are expressed in the URL—which is not impacted whether we use `RestTemplate`, `RestClient`, or `WebClient`.

Example URL with Path and Query Params

```
http://localhost:8080/rpc/greeter/say/hello?name=jim
```

A way to build a URL through type-safe convenience methods is with the `UriComponentsBuilder` class. In the following example:

- `fromHttpUrl()` - starts the URI using a string *Example Client Code Forming URL with Path and containing the base (e.g. Query Params*
- `http://localhost:8080/rpc/greeter)`
- `path()` - can be used to tack on a path to the end of the `baseUrl`. `replacePath()` is also a convenient method here to use when the value you have is the full path. Note the placeholder with `{greeting}` reserving a spot in the path. The position in the URI is important, but there is no direct relationship between what the client and service use for this placeholder name—if they use one at all.
- `queryParam()` - is used to express individual query parameters. The name of the query parameter must match what is expected by the service. Note that a placeholder was used here to express the value.
- `build()` - is used to finish off the URI. We pass in the placeholder values in the order they appear in the URI expression

```
@Test
public void say_greeting() {
    //given - a service available to
    provide a greeting
    URI url = UriComponentsBuilder
        .fromHttpUrl(baseUrl)
        .path("/say/{greeting}") ①
        .queryParam("name", "{name}") ②
        .build("hello", "jim"); ③
```

- ① path is being expressed using a `{greeting}` placeholder for the value
- ② query parameter expressed using a `{name}` placeholder for the value
- ③ values for greeting and name are filled in during call to `build()` to complete the URI

149.3. Spring HTTP Interface Parameter Handling

We can address parameters in Spring HTTP Interface using the same `@PathVariable` and `RequestParam` declarations that were used on the server-side. The following example shows making each of the parameters required. Notice also that we can have the call return the `ResponseEntity` wrapper versus just the value.

Using Required Client-side Parameter Values

```
@GetExchange("/say/{greeting}")
ResponseEntity<String> sayGreeting(
    @PathVariable(value = "greeting", required = true) String greeting,
    @RequestParam(value = "name", required=true) String name);
```

With the method defined, we can call it like a normal Java method and inspect the response.

```
//when - asking for that greeting with required parameters
... = greeterAPI.sayGreeting("hello", "jim");
//response "hello, jim"
```

149.3.1. Optional Parameters

We can make parameters optional, allowing the client to null them out. The following example shows the client passing in a null for the name—to have it defaulted by either the client or server-side code.

Optional Query Parameter Call

```
//when - asking for that greeting using client-side or server-side defaults
... = greeterAPI.sayGreeting("hello", null);
```

The optional parameter can be resolved:

- on the server-side. In this case, the client marks the parameter as not required.

Using Server-side Default Parameter Value

```
@RequestParam(value = "name", required=false) String name;
//response "hello, you"
```

- on the client-side. In this case, the client identifies the default value to use.

Using Client-side default

```
@RequestParam(value = "name", defaultValue="client") String name;
//response "hello, client"
```

Chapter 150. Accessing HTTP Responses

The target of an HTTP response may be a specific marshalled object or successful status. However, it is common to want to have access to more detailed information. For example:

- Success — was it a 201/CREATED or a 200/OK?
- Failure — was it a 400/BAD_REQUEST, 404/NOT_FOUND, 422/UNPROCESSABLE_ENTITY, or 500/INTERNAL_SERVER_ERROR?

Spring can supply that additional information in a `ResponseEntity<T>`, supplying us with:

- status code
- response headers
- response body — which will be unmarshalled to the specified type of `T`

To obtain that object — we need to adjust our call to the client.

150.1. Obtaining ResponseEntity

The client libraries offer additional calls to obtain the `ResponseEntity`.

Example RestTemplate ResponseEntity<T> Call

```
//when - asking for that greeting
ResponseEntity<String> response = restTemplate.getForEntity(url, String.class);
```

Example RestClient ResponseEntity<T> Call

```
//when - asking for that greeting
ResponseEntity<String> response = restClient.get()
    .uri(url)
    .retrieve()
    .toEntity(String.class);
```

Example WebClient ResponseEntity<T> Call

```
//when - asking for that greeting
ResponseEntity<String> response = webClient.get()
    .uri(url)
    .retrieve()
    .toEntity(String.class)
    .block();
```

Example Spring HTTP Interface Call

```
//when - asking for that greeting
```

```
 ResponseEntity<String> response = greeterAPI.sayGreeting("hello", "jim");
```

150.2. ResponseEntity<T>

The `ResponseEntity<T>` can provide us with more detail than just the response object from the body. As you can see from the following evaluation block, the client also has access to the status code and headers.

Example Returned ResponseEntity<T>

```
//then - response be successful with expected greeting
then(response.getStatusCode()).isEqualTo(HttpStatus.OK);
then(response.getHeaders().getFirst(HttpHeaders.CONTENT_TYPE)).startsWith("text/plain");
then(response.getBody()).isEqualTo("hello, jim");
```

Chapter 151. Client Error Handling

As indicated earlier, something could fail in the call to the service and we do not get our expected response returned.

Example Response Error

```
$ curl -v http://localhost:8080/rpc/greeter/boom
...
< HTTP/1.1 400
< Content-Type: application/json
< Transfer-Encoding: chunked
< Date: Thu, 21 May 2020 19:37:42 GMT
< Connection: close
<
{"timestamp":"2020-05-21T19:37:42.261+0000","status":400,"error":"Bad Request",
 "message":"Required String parameter 'value' is not present" ①
...
}
```

① Spring MVC has default error handling that will, by default return an application/json rendering of an error

Although there are differences in their options—`RestTemplate`, `RestClient`, and `WebClient` will throw an exception if the status code is not successful. Although very similar—unfortunately, `WebClient` exceptions are technically different than the others and would need separate exception handling logic if used together.

151.1. RestTemplate Response Exceptions

`RestTemplate` and `RestClient` will throw an exception, by default for error responses.

151.1.1. Default RestTemplate Exceptions

All non-`WebClient` exceptions thrown extend `HttpClientErrorException`—which is a `RuntimeException`, so handling the exception is not mandated by the Java language. The example below is catching a specific `BadRequest` exception (if thrown) and then handling the exception in a generic way.

Example RestTemplate Exception

```
import org.springframework.web.client.HttpClientErrorException;
...
//when - calling the service
HttpClientErrorException ex = catchThrowableOfType( ①
    ()->restTemplate.getEntity(url, String.class),
    HttpClientErrorException.BadRequest.class);

//when - calling the service
HttpClientErrorException ex = catchThrowableOfType(
```

```
() -> restClient.get().uri(url).retrieve().toEntity(String.class),  
HttpClientErrorException.BadRequest.class);
```

- ① using assertj `catchThrowableOfType()` to catch the exception and test that it be of a specific type only if thrown



catchThrowableOfType does not fail if no exception thrown

AssertJ `catchThrowableOfType` only fails if an exception of the wrong type is thrown. It will return a null if no exception is thrown. That allows for a "BDD style" of testing where the "when" processing is separate from the "then" verifications.

151.1.2. Noop RestTemplate Exceptions

`RestTemplate` is the only client option that allows one to bypass the exception rule and obtain an error `ResponseEntity` from the call without exception handling. The following example shows a `NoOpResponseErrorHandler` error handler being put in place and the caller is receiving the error `ResponseEntity` without using exception handling.

Example Bypass Exceptions

```
//configure RestTemplate to return error responses, not exceptions  
RestTemplate noExceptionRestTemplate = new RestTemplate();  
noExceptionRestTemplate.setErrorHandler(new NoOpResponseErrorHandler());  
  
//when - calling the service  
Assertions.assertDoesNotThrow(()->{  
    ResponseEntity<String> response = noExceptionRestTemplate.getForEntity(url,  
    String.class);  
    //then - we get a bad request  
    then(response.getStatusCode()).isEqualTo(HttpStatus.BAD_REQUEST);  
    then(response.getHeaders().getFirst(HttpHeaders.CONTENT_TYPE))  
        .isEqualTo(MediaType.APPLICATION_JSON_VALUE);  
}, "return response, not exception");
```

151.2. RestClient Response Exceptions

`RestClient` has two primary paths to invoke a request: `retrieve()` and `exchange()`.

151.2.1. RestClient retrieve() and Exceptions

`retrieve().toEntity(T)` works very similar to `RestTemplate.<method>ForEntity()`—where it returns what you ask or throws an exception. The following shows a case where the `RestClient` call will be receiving an error and throwing a `BadRequest` exception.

Default RestClient Exception Behavior with retrieve().toEntity()

```
HttpClientErrorException ex = catchThrowableOfType(  
    () -> restClient.get().uri(url).retrieve().toEntity(String.class),
```

```
HttpClientErrorException.BadRequest.class);
```

151.2.2. RestClient exchange() method

`exchange()` permits some analysis and handling of the response within the pipeline. However, it ultimately places you in a position that you need to throw an exception if you cannot return the type requested or a `ResponseEntity`. The following example shows an error being handled without an exception. One must be careful doing this since the error response likely will not be the data type requested in a realistic scenario.

Example Use of exchange() to bypass Exceptions

```
ResponseEntity<?> response = restClient.get().uri(url)
    .exchange((req, resp) -> {
        return ResponseEntity.status(resp.getStatusCode())
            .headers(resp.getHeaders())
            .body(StreamUtils.copyToString(resp.getBody(), Charset.defaultCharset()));
    });
then(ex.getStatusCode()).isEqualTo(HttpStatus.BAD_REQUEST);
```

All default `RestClient` exceptions thrown are identical to `RestTemplate` exceptions.

151.3. WebClient Response Exceptions

`WebClient` has the same two primary paths to invoke a request: `retrieve()` and `exchange()`. `retrieve()` works very similar to `RestTemplate.<method>ForEntity()`—where it returns what you ask or throws an exception. `exchange()` permits some analysis of the response—but ultimately places you in a position that you need to throw an exception if you cannot return the type requested. Overriding the exception handling design of these clients is not something I would recommend, and overriding the async API of the `WebClient` can be daunting. Therefore, I am just going to show the exception handling option.

The example below is catching a specific `BadRequest` exception and then handling the exception in a generic way.

Example WebClient Exception

```
import org.springframework.web.reactive.function.client.WebClientResponseException;
...
//when - calling the service
WebClientResponseException.BadRequest ex = catchThrowableOfType(
    () -> webClient.get().uri(url).retrieve().toEntity(String.class).block(),
    WebClientResponseException.BadRequest.class);
```

All default `WebClient` exceptions extend `WebClientResponseException`—which is also a `RuntimeException`, so it has that in common with the exception handling of `RestTemplate`.

151.4. Spring HTTP Interface Exceptions

The Spring HTTP Interface API exceptions will be identical to `RestTemplate` and `RestClient`. Any special handling of error responses can be done in the client error handling stack (e.g., `RestClient.defaultStatusHandler`). That will provide a means to translate the HTTP error response into a business exception if desired.

Example Spring HTTP Interface Exception

```
//when - calling the service
RestClientResponseException ex = catchThrowableOfType(
    () -> greeterAPI.boom(),
    HttpClientErrorException.BadRequest.class);
```

151.5. Client Exceptions

Once the code calling one of the two clients has the client-specific exception object, they have access to three key response values:

- HTTP status code
- HTTP response headers
- HTTP body as string or byte array

The following is an example of handling an exception thrown by `RestTemplate` and `RestClient`.

Example RestTemplate/RestClient Exception Inspection

```
HttpClientErrorException ex = ...

//then - we get a bad request
then(ex.getStatusCode()).isEqualTo(HttpStatus.BAD_REQUEST);
then(ex.getResponseHeaders().getFirst(HttpHeaders.CONTENT_TYPE))
    .isEqualTo(MediaType.APPLICATION_JSON_VALUE);
log.info("{}" , ex.getResponseBodyAsString());
```

The following is an example of handling an exception thrown by `WebClient`.

Example WebClient Exception Inspection

```
WebClientResponseException.BadRequest ex = ...

//then - we get a bad request
then(ex.getStatusCode()).isEqualTo(HttpStatus.BAD_REQUEST);
then(ex.getHeaders().getFirst(HttpHeaders.CONTENT_TYPE)) ①
    .isEqualTo(MediaType.APPLICATION_JSON_VALUE);
log.info("{}" , ex.getResponseBodyAsString());
```

① `WebClient`'s exception method name to retrieve response headers different from `RestTemplate`

Chapter 152. Controller Responses

In our earlier example, our only response option from the service was a limited set of status codes derived by the container based on what was returned. The specific error demonstrated was generated by the Spring MVC container based on our mapping definition. It will be common for the controller method itself to need explicit control over the HTTP response returned --primarily to express response-specific

- HTTP status code
- HTTP headers

152.1. Controller Return ResponseEntity

The following service example performs some trivial error checking and:

- responds with an explicit error if there is a problem with the input
- responds with an explicit status and Content-Location header if successful

The service provides control over the entire response by returning a `ResponseType` containing the complete HTTP result versus just returning the result value for the body. The `ResponseType` can express status code, headers, and the returned entity.

Example Controller Returning ResponseEntity

```
import org.springframework.web.servlet.support.ServletUriComponentsBuilder;
...
    @RequestMapping(path="boys",
        method=RequestMethod.GET)
    public ResponseEntity<String> createBoy(@RequestParam("name") String name) { ①
        try {
            someMethodThatMayThrowException(name);

            String url = ServletUriComponentsBuilder.fromCurrentRequest() ②
                .build().toUriString();
            return ResponseEntity.ok() ③
                .header(HttpHeaders.CONTENT_LOCATION, url)
                .body(String.format("hello %s, how do you do?", name));
        } catch (IllegalArgumentException ex) {
            return ResponseEntity.unprocessableEntity() ④
                .body(ex.toString());
        }
    }
    private void someMethodThatMayThrowException(String name) {
        if ("blue".equalsIgnoreCase(name)) {
            throw new IllegalArgumentException("boy named blue?");
        }
    }
}
```

- ① `ResponseEntity` returned used to express full HTTP response
- ② `ServletUriComponentsBuilder` is a URI builder that can provide context of current call
- ③ service is able to return an explicit HTTP response with appropriate success details
- ④ service is able to return an explicit HTTP response with appropriate error details

152.2. Example ResponseEntity Responses

In response, we see the explicitly assigned status code and Content-Location header.

Example ResponseEntity Success Returned

```
curl -v http://localhost:8080/rpc/greeter/boys?name=jim
...
< HTTP/1.1 200 ①
< Content-Location: http://localhost:8080/rpc/greeter/boys?name=jim ②
< Content-Type: text/plain;charset=UTF-8
< Content-Length: 25
...
hello jim, how do you do?
```

- ① status explicitly
- ② Content-Location header explicitly supplied by service

For the error condition, we see the explicit status code and error payload assigned.

Example ResponseEntity Error Returned

```
$ curl -v http://localhost:8080/rpc/greeter/boys?name=blue
...
< HTTP/1.1 422 ①
< Content-Type: text/plain;charset=UTF-8
< Content-Length: 15
...
boy named blue?
```

- ① HTTP status code explicitly supplied by service

152.3. Controller Exception Handler

We can make a small but significant step at simplifying the controller method by making sure the exception thrown is fully descriptive and moving the exception handling to either:

- a separate, annotated method of the controller or
- globally to be used by all controllers (shown later).

The following example uses `@ExceptionHandler` annotation to register a handler for when controller methods happen to throw the `IllegalArgumentException`. The handler can return an explicit

ResponseEntity with the error details.

Example Controller ExceptionHandler

```
import org.springframework.web.bind.annotation.ExceptionHandler;  
...  
@ExceptionHandler(IllegalArgumentException.class) ①  
public ResponseEntity<String> handle(IllegalArgumentException ex) {②  
    return ResponseEntity.unprocessableEntity() ③  
        .body(ex.getMessage());  
}
```

① handles all `IllegalArgumentException`-s thrown by controller method (or anything it calls)

② input parameter is concrete type or parent type of handled exception

③ handler builds a `ResponseEntity` with the details of the error



Create custom exceptions to address specific errors

Create custom exceptions to the point that the handler has the information and context it needs to return a valuable response.

152.4. Simplified Controller Using ExceptionHandler

With all exceptions addressed by `ExceptionHandlers`, we can free our controller methods of tedious, repetitive conditional error reporting logic and still return an explicit HTTP response.

Example Controller Method using ExceptionHandler

```
@RequestMapping(path="boys/throws",  
    method=RequestMethod.GET)  
public ResponseEntity<String> createBoyThrows(@RequestParam("name") String name) {  
    someMethodThatMayThrowException(name); ①  
  
    String url = ServletUriComponentsBuilder.fromCurrentRequest()  
        .replacePath("/rpc/greeter/boys") ②  
        .build().toUriString();  
  
    return ResponseEntity.ok()  
        .header(HttpHeaders.CONTENT_LOCATION, url)  
        .body(String.format("hello %s, how do you do?", name));  
}
```

① Controller method is free from dealing with exception logic

② replacing a path to match sibling implementation response

Note the new method endpoint with the exception handler returns the same, explicit HTTP response as the earlier example.

Example ExceptionHandler Response

```
curl -v http://localhost:8080/rpc/greeter/boys/throws?name=blue
...
< HTTP/1.1 422
< Content-Type: text/plain;charset=UTF-8
< Content-Length: 15
...
boy named blue?
```

Chapter 153. Summary

In this module we:

- identified two primary paradigms (synchronous and reactive) and web frameworks (Spring MVC and Spring WebFlux) for implementing web processing and communication
- implemented an HTTP endpoint for a URI and method using Spring MVC annotated controller in a fully synchronous mode
- demonstrated how to pass parameters between client and service using path and query parameters
- demonstrated how to pass return results from service to client using http status code, response headers, and response body
- demonstrated how to explicitly set HTTP responses in the service
- demonstrated how to clean up service logic by using exception handlers
- demonstrated use of the synchronous Spring MVC `RestTemplate` and `RestClient` and reactive Spring WebFlux `WebClient` client APIs
- demonstrated use of Spring HTTP Interface to wrap low-level client APIs with a type-safe, business interface

Controller/Service Interface

copyright © 2024 jim stafford (jim.stafford@jhu.edu)

Chapter 154. Introduction

Many times we may think of a service from the client's perspective and term everything on the other side of the HTTP connection to be "the service". That is OK from the client's perspective, but even a moderately-sized service—there are normally a few layers of classes playing a certain architectural role and that front-line controller we have been working with should primarily be a "web facade" interfacing the business logic to the outside world.

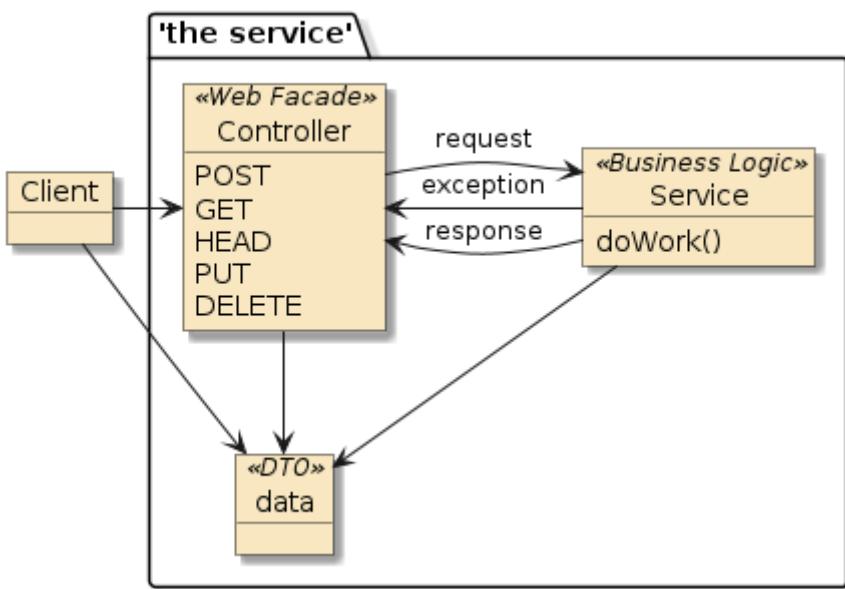


Figure 49. Controller/Service Relationship

In this lecture we are going to look more closely at how the overall endpoint breaks down into a set of "facade" and "business logic" pattern players and lay the groundwork for the "Data Transfer Object" (DTO) covered in the next lecture.

154.1. Goals

The student will learn to:

- identify the Controller class as having the role of a facade
- encapsulate business logic within a separate service class
- establish some interface patterns between the two layers so that the web facade is as clean as possible

154.2. Objectives

At the conclusion of this lecture and related exercises, the student will be able to:

1. implement a service class to encapsulate business logic
2. turn `@RestController` class into a facade and delegate business logic details to an injected service class
3. identify error reporting strategy options

4. identify exception design options
5. implement a set of condition-specific exceptions
6. implement a Spring `@RestControllerAdvice` class to offload exception handling and error reporting from the `@RestController`

Chapter 155. Roles

In an N-tier, distributed architecture there is commonly a set of patterns to apply to our class design.

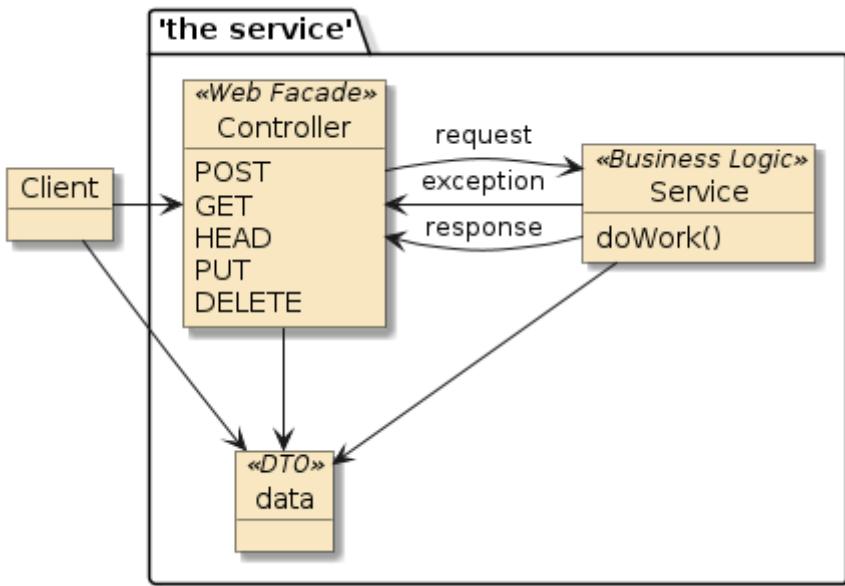


Figure 50. Controller/Service Relationship

- **Business Logic** - primary entry point for doing work. The business logic knows the why and when to do things. Within the overall service—this is the class (or set of classes) that make up the core service.
- **Data Transfer Object (DTO)** - used to describe requested work to the business logic or results from the business logic. In small systems, the DTO may be the same as the business objects (BO) stored in the database—but the specific role that will be addressed here is communicating outside of the overall service.
- **Facade** - this provides an adapter around the business logic that translates commands from various protocols and libraries—into core language commands.

I will cover DTOs in more detail in the next lecture—but relative to the client, facade, and business logic—know that all three work on the same type of data. The DTO data types pass thru the controller without a need for translation—other than what is required for communications.

Our focus in this lecture is still the controller and will now look at some controller/service interface strategies that will help develop a clean web facade in our controller classes.

Chapter 156. Error Reporting

When an error occurs—whether it be client or internal server errors—we want to have access to useful information that can be used to correct or avoid the error in the future. For example, if a client asks for information on a particular account that cannot be found, it would save minutes to hours of debugging to know whether the client requested a valid account# or the bank's account repository was not currently available.

We have one of two techniques to report error details: complex object result and thrown exception.

Design a way to allow low-level code report context of failures



The place where the error is detected is normally the place with the most amount of context details known. Design a way to have the information from the detection spot propagated to the error handling.

156.1. Complex Object Result

For the complex object result approach, each service method returns a complex result object (similar in concept to `ResponseEntity`). If the business method is:

- **successful:** the requested result is returned
- **unsuccessful:** the returned result expresses the error

The returned method type is complex enough to carry both types of payloads.

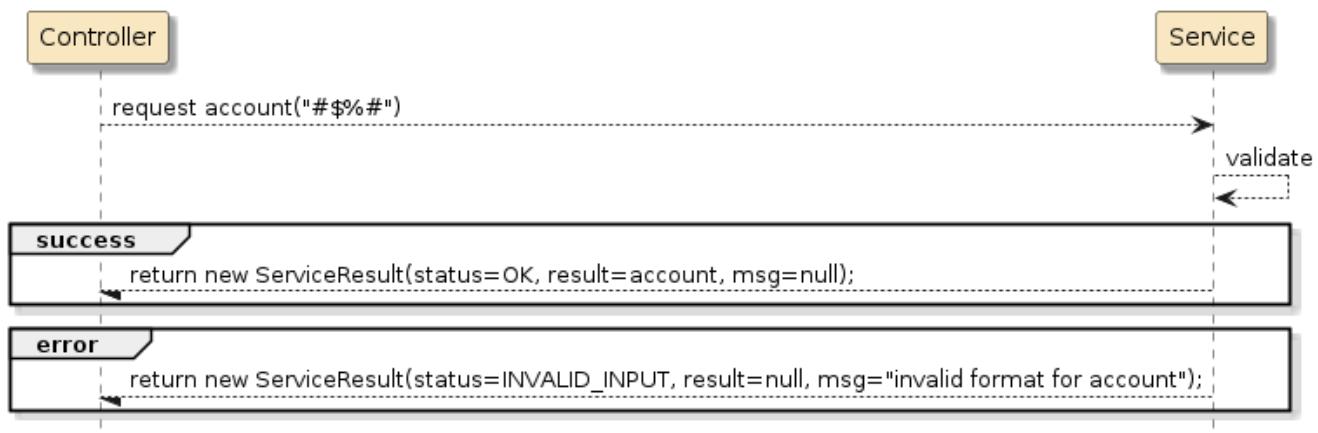


Figure 51. Service Returns Complex Object with Status and Error

Complex return objects require handling logic in caller



The complex result object requires the caller to have error handling logic ready to triage and react to the various responses. Anything that is not immediately handled may accidentally be forgotten.

156.2. Thrown Exception

For the thrown exception case, exceptions are declared to carry failure-specific error reporting. The

business method only needs to declare the happy path response in the return of the method and optionally declare try/catch blocks for errors it can address.

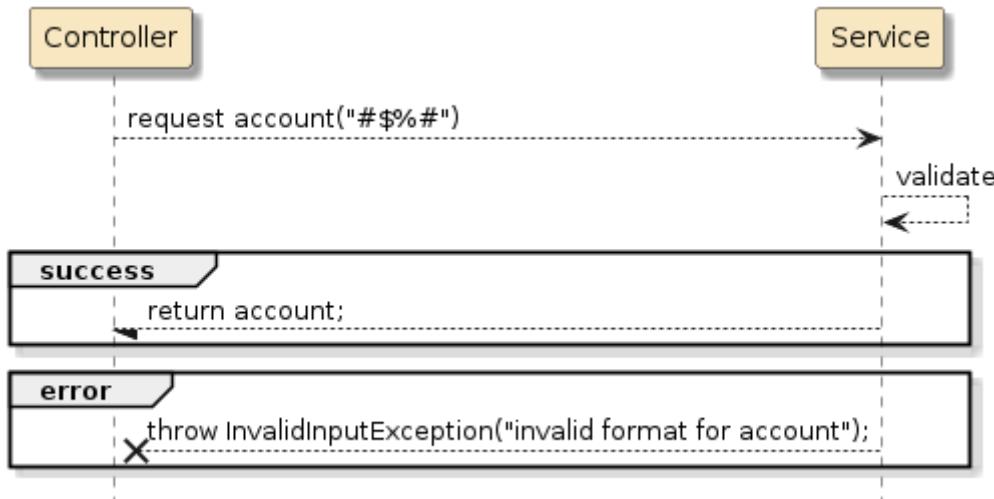


Figure 52. Service Throws Exception of Type of Error



Thrown exceptions give the caller the option to handle or delegate

The thrown exception technique gives the caller the option to construct a try/catch block and immediately handle the error or to automatically let it propagate to a caller that can address the issue.

Either technique will functionally work. However, returning the complex object versus exception will require manual triage logic on the receiving end. As long as we can create error-specific exceptions, we can create some cleaner handling options in the controller.

156.3. Exceptions

Going the exception route, we can start to consider:

- what specific errors should our services report?
- what information is reported?
 - timestamp
 - (descriptive? redacted?) error text
- are there generalizations or specializations?

The HTTP [organization of status codes](#) is a good place to start thinking of error types and how to group them (i.e., it is used by the world's largest information system—the WWW). HTTP defines two primary types of errors:

- client-based
- server-based

It could be convenient to group them into a single hierarchy—depending on how we defined the details of the exceptions.

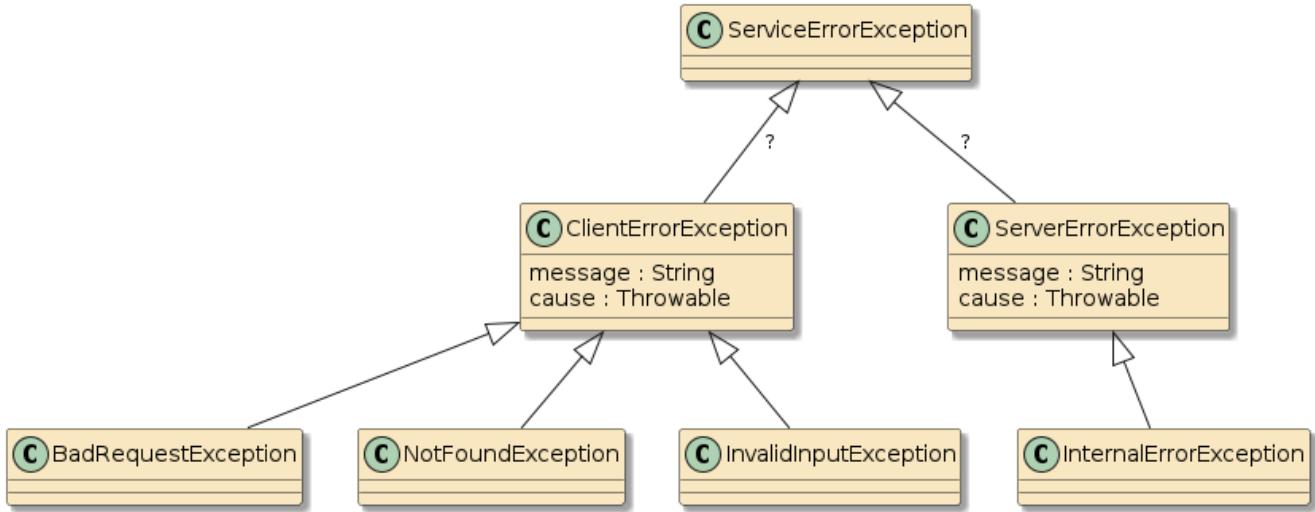


Figure 53. Example Service Exception Hierarchy

From the start, we can easily guess that our service method(s) might fail because

- **NotFoundException**: the target entity was not found
- **InvalidInputException**: something wrong with the content of what was requested
- **BadRequestException**: request was not understood or erroneously requested
- **InternalErrorException**: infrastructure or something else internal went bad

We can also assume that we would need, at a minimum

- a message - this would ideally include IDs that are specific to the context
- cause exception - commonly something wrapped by a server error

156.4. Checked or Unchecked?

Going the exception route—the most significant impact to our codebase will be the choice of checked versus unchecked exceptions (i.e., `RuntimeException`).

- **Checked Exception** - these exceptions inherit from `java.lang.Exception` and are required to be handled by a try/catch block or declared as rethrown by the calling method. It always starts off looking like a good practice, but can get quite tedious when building layers of methods.
- **RuntimeException** - these exceptions inherit from `java.lang.RuntimeException` and not required to be handled by the calling method. This can be a convenient way to address exceptions "not dealt with here". However, it is always the caller's option to catch any exception they can specifically address.

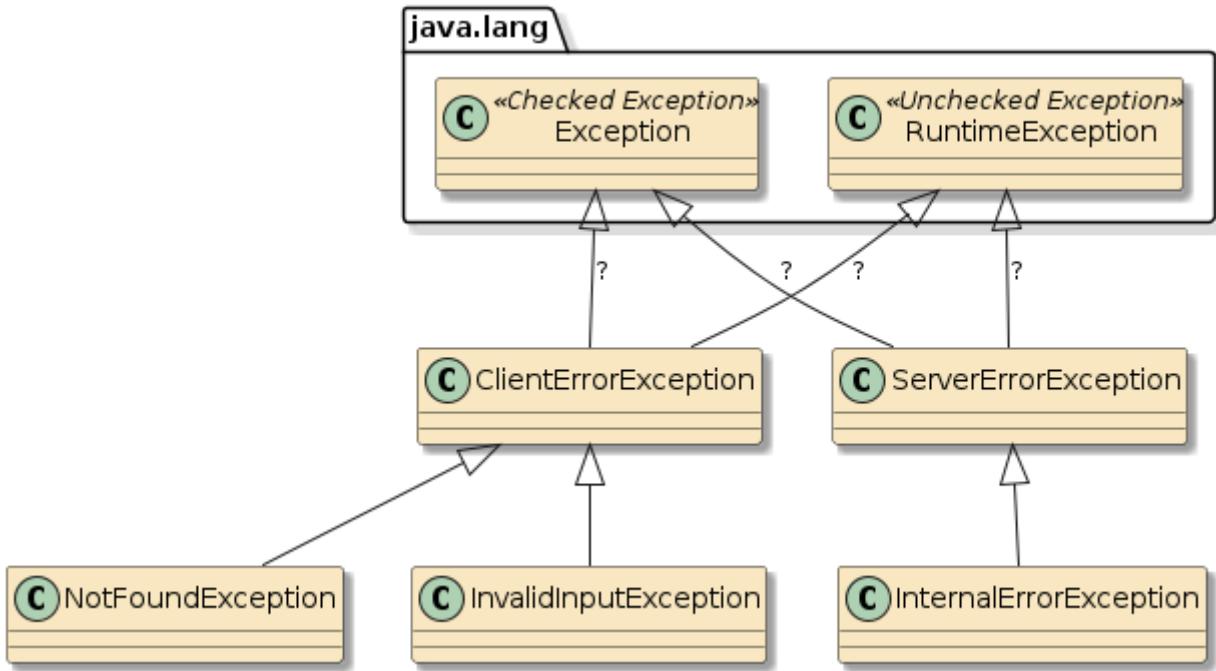


Figure 54. Should Reported Exceptions be Checked or Unchecked?

If we choose to make them different (i.e., `ServerErrorException` unchecked and `ClientErrorException` checked), we will have to create separate inheritance hierarchies (i.e., no common `ServiceException` parent).

156.5. Candidate Client Exceptions

The following is a candidate implementation for client exceptions. I am going to go the seemingly easy route and make them unchecked/`RuntimeExceptions`—but keep them in a separate hierarchy from the server exceptions to allow an easy change. Complete examples can be located in the [repository](#)

Candidate Client Exceptions

```

public abstract class ClientErrorException extends RuntimeException {
    protected ClientErrorException(Throwable cause) {
        super(cause);
    }
    protected ClientErrorException(String message, Object...args) {
        super(String.format(message, args)); ①
    }
    protected ClientErrorException(Throwable cause, String message, Object...args) {
        super(String.format(message, args), cause);
    }

    public static class NotFoundException extends ClientErrorException {
        public NotFoundException(String message, Object...args)
            { super(message, args); }
        public NotFoundException(Throwable cause, String message, Object...args)
            { super(cause, message, args); }
    }
}

```

```

public static class InvalidInputException extends ClientErrorException {
    public InvalidInputException(String message, Object...args)
        { super(message, args); }
    public InvalidInputException(Throwable cause, String message, Object...args)
        { super(cause, message, args); }
}

```

- ① encourage callers to add instance details to exception by supplying built-in, optional formatter

The following is an example of how the caller can instantiate and throw the exception based on conditions detected in the request.

Example Client Exception Throw

```

if (null==gesture) {
    throw new ClientErrorException
        .NotFoundException("gesture type[%s] not found", gestureType);
}

```

156.6. Service Errors

The following is a candidate implementation for server exceptions. These types of errors are commonly unchecked.

```

public abstract class ServerErrorException extends RuntimeException {
    protected ServerErrorException(Throwable cause) {
        super(cause);
    }
    protected ServerErrorException(String message, Object...args) {
        super(String.format(message, args));
    }
    protected ServerErrorException(Throwable cause, String message, Object...args) {
        super(String.format(message, args), cause);
    }

    public static class InternalErrorException extends ServerErrorException {
        public InternalErrorException(String message, Object...args)
            { super(message, args); }
        public InternalErrorException(Throwable cause, String message, Object...args)
            { super(cause, message, args); }
    }
}

```

The following is an example of instantiating and throwing a server exception based on a caught exception.

Example Server Exception Throw

```
try {  
    //...  
} catch (RuntimeException ex) {  
    throw new InternalErrorException(ex, ①  
        "unexpected error getting gesture[%s]", gestureType); ②  
}
```

① reporting source exception forward

② encourage callers to add instance details to exception by supplying built-in, optional formatter

Chapter 157. Controller Exception Advice

We saw earlier where we could register an exception handler within the controller class and how that could clean up our controller methods of noisy error handling code. I want to now build on that concept and our new concrete service exceptions to define an external controller advice that will handle all registered exceptions.

The following is an example of a controller method that is void of error handling logic because of the external controller advice we will put in place.

Example Controller Method - Void of Error Handling

```
@RestController
public class GesturesController {
    ...
    @RequestMapping(path=GESTURE_PATH,
                    method=RequestMethod.GET,
                    produces = {MediaType.TEXT_PLAIN_VALUE})
    public ResponseEntity<String> getGesture(
        @PathVariable(name="gestureType") String gestureType,
        @RequestParam(name="target", required=false) String target) {
        //business method
        String result = gestures.getGesture(gestureType, target); ①

        String location = ServletUriComponentsBuilder.fromCurrentRequest()
            .build().toUriString();
        return ResponseEntity
            .status(HttpStatus.OK)
            .header(HttpHeaders.CONTENT_LOCATION, location)
            .body(result);
    }
}
```

① handles only successful result—exceptions left to controller advice

157.1. Service Method with Exception Logic

The following is a more complete example of the business method within the service class. Based on the result of the interaction with the data access tier—the business method determines the gesture does not exist and reports that error using an exception.

Example Service Method with Exception Error Reporting Logic

```
@Service
public class GesturesServiceImpl implements GesturesService {
    @Override
    public String getGesture(String gestureType, String target) {
        String gesture = gestures.get(gestureType); //data access method
        if (null==gesture) {
            throw new ClientErrorException ①
        }
    }
}
```

```

        .NotFoundException("gesture type[%s] not found", gestureType);
    } else {
        String response = gesture + (target==null ? "" : ", " + target);
        return response;
    }
}
...

```

① service reporting details of error

157.2. Controller Advice Class

The following is a controller advice class. We annotate this with `@RestControllerAdvice` to better describe its role and give us the option to create fully annotated handler methods.

My candidate controller advice class contains an internal helper method that programmatically builds a `ResponseEntity`. The type-specific exception handler must translate the specific exception into a HTTP status code and body. A more complete example—designed to be a base class to concrete `@RestControllerAdvice` classes—can be found in the [repository](#).

Controller Advice Class

```

package info.ejava.examples.svc.httpapi.gestures.controllers;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.RestControllerAdvice;

@RestControllerAdvice( ①
// wraps ==> @ControllerAdvice
//           wraps ==> @Component
    basePackageClasses = GesturesController.class) ②
public class ExceptionAdvice { /③
    //internal helper method
    protected ResponseEntity<String> buildResponse(HttpStatus status, ④
                                                       String text) { ⑤

        return ResponseEntity
            .status(status)
            .body(text);
    }
}
...

```

① `@RestControllerAdvice` denotes this class as a `@Component` that will handle thrown exceptions

② optional annotations can be used to limit the scope of this advice to certain packages and controller classes

③ handled thrown exceptions will return the DTO type for this application—in this case just `text/plain`

④ type-specific exception handlers must map exception to an HTTP status code

⑤ type-specific exception handlers must produce error text

Example assumes DTO type is plain/test string



This example assumes the DTO type for errors is a `text/plain` string. More robust response type would be part of an example using complex DTO types.

157.3. Advice Exception Handlers

Below are the candidate type-specific exception handlers we can use to translate the context-specific information from the exception to a valuable HTTP response to the client.

Advice Exception Handlers

```
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ExceptionHandler;

import static info.ejava.examples.svc.httpapi.gestures.svc.ClientErrorException.*;
import static info.ejava.examples.svc.httpapi.gestures.svc.ServerErrorException.*;

...
@ExceptionHandler(NotFoundException.class) ①
public ResponseEntity<String> handle(NotFoundException ex) {
    return buildResponse(HttpStatus.NOT_FOUND, ex.getMessage()); ②
}
@ExceptionHandler(InvalidInputException.class)
public ResponseEntity<String> handle(InvalidInputException ex) {
    return buildResponse(HttpStatus.UNPROCESSABLE_ENTITY, ex.getMessage());
}
@ExceptionHandler(InternalErrorException.class)
public ResponseEntity<String> handle(InternalErrorException ex) {
    log.warn("{}\n", ex.getMessage(), ex); ③
    return buildResponse(HttpStatus.INTERNAL_SERVER_ERROR, ex.getMessage());
}
@ExceptionHandler(RuntimeException.class)
public ResponseEntity<String> handleRuntimeException(RuntimeException ex) {
    log.warn("{}\n", ex.getMessage(), ex); ③
    String text = String.format(
        "unexpected error executing request: %s", ex.toString());
    return buildResponse(HttpStatus.INTERNAL_SERVER_ERROR, text);
}
```

① annotation maps the handler method to a thrown exception type

② handler method receives exception and converts to a `ResponseEntity` to be returned

③ the unknown error exceptions are candidates for mandatory logging

Chapter 158. Summary

In this module we:

- identified the `@RestController` class' role is a "facade" for a web interface
- encapsulated business logic in a `@Service` class
- identified data passing between clients, facades, and business logic is called a Data Transfer Object (DTO). The DTO was a string in this simple example, but will be expanded in the content lecture
- identified how exceptions could help separate successful business logic results from error path handling
- identified some design choices for our exceptions
- identified how a controller advice class can be used to offload exception handling

API Data Formats

copyright © 2024 jim stafford (jim.stafford@jhu.edu)

Chapter 159. Introduction

Web content is shared using many standardized [MIME Types](#). We will be addressing two of them here

- XML
- JSON

I will show manual approaches to marshaling/unmarshalling first. However, content is automatically marshalled/unmarshalled by the web client container once everything is set up properly. Manual marshaling/unmarshalling approaches are mainly useful in determining provider settings and annotations—as well as to perform low-level development debug outside the server on the shape and content of the payloads.

159.1. Goals

The student will learn to:

- identify common/standard information exchange content types for web API communications
- manually marshal and unmarshal Java types to and from a data stream of bytes for multiple content types
- negotiate content type when communicating using web API
- pass complex Data Transfer Objects to/from a web API using different content types
- resolve data mapping issues

159.2. Objectives

At the conclusion of this lecture and related exercises, the student will be able to:

1. design a set of Data Transfer Objects (DTOs) to render information from and to the service
2. define Java class content type mappings to customize marshalling/unmarshalling
3. specify content types consumed and produced by a controller
4. specify content types supplied and accepted by a client

Chapter 160. Pattern Data Transfer Object

There can be multiple views of the same conceptual data managed by a service. They can be the same physical implementation—but they serve different purposes that must be addressed. We will be focusing on the external client view (Data Transfer Object (DTO)) during this and other web tier lectures. I will specifically contrast the DTO with the internal implementation view (Business Object (BO)) right now to help us see the difference in the two roles.

160.1. DTO Pattern Problem Space

Context

Business Objects (data used directly by the service tier and potentially mapped directly to the database) represent too much information or behavior to transfer to remote client

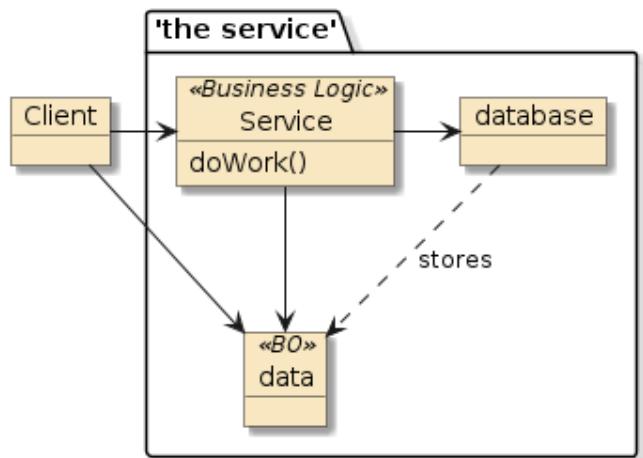


Figure 55. Clients and Service Sharing Implementation Data

Problem

Issues can arise when service implementations are complex.

- client may get data they do not need
- client may get data they cannot handle
- client may get data they are not authorized to use
- client may get too much data to be useful (e.g., entire database serialized to client)

Forces

The following issues are assumed to be true:

- some clients are local and can share object references with business logic
- handling specifics of remote clients is outside core scope of business logic

160.2. DTO Pattern Solution Space

Solution

- define a set of data that is appropriate for transferring requests and responses between client and service
 - define a Remote (Web) Facade over Business Logic to handle remote communications with the client
 - remote Facade constructs Data Transfer Objects (DTOs) from Business Objects that are appropriate for remote client view
 - remote Facade uses DTOs to construct or locate Business Objects to communicate with Business Logic

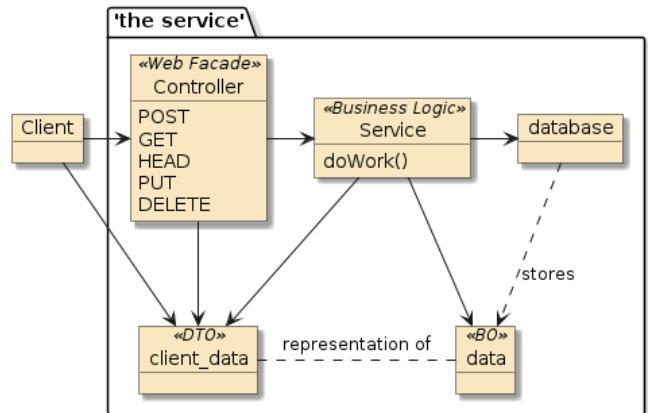


Figure 56. DTO Represents Client View of Data

DTO/BO Mapping Location is a Design Choice

The design decision of which layer translates between DTOs of the API and BOs of the service is not always fixed. Since the DTO is an interface pattern and the Web API is one of many possible interface facades and clients of the service — the job of DTO/BO mapping may be done in the service tier instead.

160.3. DTO Pattern Players

Data Transfer Object

- represents a subset of the state of the application at a point in time
 - not dependent on Business Objects or server-side technologies
 - doing so would require sending Business Objects to client
 - XML and JSON provide the “ultimate isolation” in DTO implementation/isolation

Remote (Web) Facade

- uses Business Logic and DTOs to perform core business logic
 - manages interface details with client

Business Logic

- performs core implementation duties that may include interaction with backend services and databases

Business Object (Entity)

- representation of data required to implement service
 - may have more server-side-specific logic when DTOs are present in the design

DTOs and BOs can be same class(es) in simple or short-lived services

DTOs and BOs can be the same class in small services. However, supporting multiple versions of clients over longer service lifetimes may cause even small services to split the two data models into separate implementations.

Chapter 161. Sample DTO Class

The following is an example DTO class we will look to use to represent client view of data in a simple "Quote Service". The `QuoteDTO` class can start off as a simple POJO and—depending on the binding (e.g., JSON or XML) and binding library (e.g., Jackson, JSON-B, or JAXB) - we may have to add external configuration and annotations to properly shape our information exchange.

The class is a vanilla POJO with a default constructor, public getters and setters, and other convenience methods—mostly implemented by Lombok. The quote contains three different types of fields (int, String, and LocalDate). The `date` field is represented using `java.time.LocalDate`.

Example Starting POJO for DTO

```
package info.ejava.examples.svc.content.quotes.dto;

import lombok.*;
import java.time.LocalDate;

@Data
@Builder
@With
@NoArgsConstructor ①
@AllArgsConstructor
public class QuoteDTO {
    private int id;
    private String author;
    private String text;
    private LocalDate date; ③
    private String ignored; ④
}
```

① default constructor

② public setters and getters

③ using Java 8, `java.time.LocalDate` to represent generic day of year without timezone

④ example attribute we will configure to be ignored

Lombok @Builder and @With



`@Builder` will create a new instance of the class using incrementally defined properties. `@With` creates a copy of the object with a new value for one of its properties. `@Builder` can be configured to create a copy constructor (i.e., a copy builder with no property value change).

Lombok @Builder and Constructors



`@Builder` requires an all-args-ctor and will define a package-friendly one unless there is already a ctor defined. Unmarshallers require a no-args-ctor and can be provided using `@NoArgsConstructor`. The presence of the no-args-ctor turns off the required all-args-ctor for `@Builder` and can be re-enabled with `@AllArgsConstructor`.

Chapter 162. Time/Date Detour

While we are on the topic of exchanging data — we might as well address time-related data that can cause numerous mapping issues. Our issues are on multiple fronts.

- what does our time-related property represent?
 - e.g., a point in time, a point in time in a specific timezone, a birthdate, a daily wake-up time
- what type do we use to represent our expression of time?
 - do we use legacy Date-based types that have a lot of support despite ambiguity issues?
 - do we use the newer `java.time` types that are more explicit in meaning but have not fully caught on everywhere?
- how should we express time within the marshalled DTO?
- how can we properly unmarshal the time expression into what we need?
- how can we handle the alternative time wire expressions with minimal pain?

162.1. Pre Java 8 Time

During pre-Java8, we primarily had the following time-related `java.util` classes

<code>Date</code>	represents a point in time without timezone or calendar information. The point is a Java long value that represents the number of milliseconds before or after 1970 UTC. This allows us to identify a millisecond between 292,269,055 BC and 292,278,994 AD when applied to the Gregorian calendar.
<code>Calendar</code>	interprets a Date according to an assigned calendar (e.g., Gregorian Calendar) into years, months, hours, etc. Calendar can be associated with a specific timezone offset from UTC and assumes the Date is relative to that value.

During the pre-Java 8 time period, there was also a time-based library called `Joda` that became popular at providing time expressions that more precisely identified what was being conveyed.

162.2. `java.time`

The ambiguity issues with `java.util.Date` and the expression and popularity of Joda caused it to be adopted into Java 8 ([JSR 310](#)). The following are a few of the key `java.time` constructs added in Java 8.

<code>Instant</code>	represents a point in time at 00:00 offset from UTC. The point is a nanosecond and improves on <code>java.util.Date</code> by assigning a specific UTC timezone. The <code>toString()</code> method on <code>Instant</code> will always print a UTC-relative value (<code>1970-01-01T00:00:00.000000001Z</code>).
<code>OffsetDate</code> <code>Time</code>	adds <code>Calendar</code> -like view to an Instant with a fixed timezone offset (<code>1970-01-01T00:00:00-04:00</code>).

ZonedDateTime	adds timezone identity to OffsetDateTime — which can be used to determine the appropriate timezone offset (i.e., daylight savings time) (<code>1969-12-31T23:00:00.000000001-05:00[America/New_York]</code> , <code>1769-12-31T23:03:58.000000001-04:56:02[America/New_York]</code>). This class is useful in presenting current time relative to where and when the time is represented. For example, during early testing I made a typo in my 1776 date and used 1976 for year. I also used ZoneId.systemDefault() ("America/New_York"). The ZoneId had a -04:00 hour difference from UTC in 1976 and a peculiar -04:56:02 hour difference from UTC in 1776. ZoneId has the ability to derive a different timezone offset based on rules for that zone.
LocalDate	a generic date, independent of timezone and time (<code>1970-01-01</code>). A common example of this is a birthday or anniversary.
LocalTime	a generic time of day, independent of timezone or specific date. This allows us to express "I set my alarm for 6am" - without specifying the actual dates that is performed (<code>00:00:00.000000001</code>).
LocalDateTime	a date and time but lacking a specific timezone offset from UTC (<code>1970-01-01T00:00:00.000000001</code>). This allows a precise date/time to be stored that is assumed to be at a specific timezone offset (usually UTC) — without having to continually store the timezone offset to each instance.
Duration	a time-based amount of time (e.g., 30.5 seconds). Vocabulary supports from milliseconds to days.
Period	a date based amount of time (e.g., 2 years and 1 day). Vocabulary supports from days to years.

162.3. Date/Time Formatting

There are two primary format frameworks for formatting and parsing time-related fields in text fields like XML or JSON:

java.text.DateFormat	This legacy <code>java.text</code> framework's primary job is to parse a String of text into a Java Date instance or format a Java Date instance into a String of text. Subclasses of <code>DateFormat</code> take care of the details and <code>java.text.SimpleDateFormat</code> accepts a String format specification. An example format <code>yyyy-MM-dd\T\HH:mm:ss.SSSX</code> assigned to UTC and given a Date for the 4th of July would produce <code>1776-07-04T00:00:00.000Z</code> .
java.time.format.DateTimeFormatter	This newer <code>java.time</code> formatter performs a similar role to <code>DateFormat</code> and <code>SimpleDateFormat</code> combined. It can parse a String into <code>java.time</code> constructs as well as format instances to a String of text. It does not work directly with Dates, but the <code>java.time</code> constructs it does produce can be easily converted to/from <code>java.util.Date</code> thru the <code>Instance</code> type. The coolest thing about <code>DateTimeFormatter</code> is that not only can it be configured using a parsable string — it can also be defined using Java objects. The following is an example of the formatter. It is built using the ISO_LOCAL_DATE and ISO_LOCAL_TIME formats.

Example DateTimeFormatter.ISO_LOCAL_DATE_TIME

```
public static final DateTimeFormatter ISO_LOCAL_DATE_TIME;
static {
    ISO_LOCAL_DATE_TIME = new DateTimeFormatterBuilder()
        .parseCaseInsensitive()
        .append(ISO_LOCAL_DATE)
        .appendLiteral('T')
        .append(ISO_LOCAL_TIME)
        .toFormatter(ResolverStyle.STRICT, IsoChronology.INSTANCE);
}
```

This, wrapped with some optional and default value constructs to handle missing information makes for a pretty powerful time parsing and formatting tool.

162.4. Date/Time Exchange

There are a few time standards supported by Java date/time formatting frameworks:

ISO 8601	This standard is cited in many places but hard to track down an official example of each and every format—especially when it comes to 0 values and timezone offsets. However, an example representing a ZonedDateTime and EST may look like the following: 1776-07-04T02:30:00.123456789-05:00 and 1776-07-04T07:30:00.123456789Z . The nanoseconds field is 9 digits long but can be expressed to a level of supported granularity—commonly 3 decimal places for <code>java.util.Date</code> milliseconds.
RFC 822/ RFC 1123	These are lesser followed standards for APIs and includes constructs like an English word abbreviation for day of week and month. The DateTimeFormatter example for this group is Tue, 3 Jun 2008 11:05:30 GMT ^[1]

My examples will work exclusively with the ISO 8601 formats. The following example leverages the Java expression of time formatting to allow for multiple offset expressions (`Z`, `+00`, `+0000`, and `+00:00`) on top of a standard LOCAL_DATE_TIME expression.

Example Lenient ISO Date/Time Parser

```
public static final DateTimeFormatter UNMARSHALLER
= new DateTimeFormatterBuilder()
    .parseCaseInsensitive()
    .append(DateTimeFormatter.ISO_LOCAL_DATE)
    .appendLiteral('T')
    .append(DateTimeFormatter.ISO_LOCAL_TIME)
    .parseLenient()
    .optionalStart().appendOffset("+HH", "Z").optionalEnd()
    .optionalStart().appendOffset("+HH:mm", "Z").optionalEnd()
    .optionalStart().appendOffset("+HHmm", "Z").optionalEnd()
    .optionalStart().appendLiteral('[').parseCaseSensitive()
        .appendZoneRegionId()
        .appendLiteral(']').optionalEnd()
    .parseDefaulting(ChronoField.OFFSET_SECONDS, 0)
```

```
.parseStrict()  
.toFormatter();
```

Use ISO_LOCAL_DATE_TIME Formatter by Default



Going through the details of `DateTimeFormatterBuilder` is out of scope for what we are here to cover. Using the `ISO_LOCAL_DATE_TIME` formatter should be good enough in most cases.

[1] "[DateTimeFormatter RFC_1123_DATE_TIME Javadoc](#)", DateTimeFormatter Javadoc, Oracle

Chapter 163. Java Marshallers

I will be using four different data marshalling providers during this lecture:

- **Jackson JSON** the default JSON provider included within Spring and Spring Boot. It implements its own proprietary interface for mapping Java POJOs to JSON text.
- **JSON Binding** a relatively new Jakarta EE standard for JSON marshalling. The reference implementation is [Yasson](#) from the open source Glassfish project. It will be used to verify and demonstrate portability between the built-in Jackson JSON and other providers.
- **Jackson XML** a tightly integrated sibling of Jackson JSON. This requires a few extra module dependencies but offers a very similar setup and annotation set as the JSON alternative. I will use Jackson XML as my primary XML provider during examples.
- **Java Architecture for XML Binding (JAXB)** a well-seasoned XML marshalling framework that was the foundational requirement for early JavaEE servlet containers. I will use JAXB to verify and demonstrate portability between Jackson XML and other providers.

Spring Boot comes with a Jackson JSON pre-wired with the web dependencies. It seamlessly gets called from RestTemplate, RestClient, WebClient and the RestController when [application/json](#) or nothing has been selected. Jackson XML requires additional dependencies — but integrates just as seamlessly with the client and server-side frameworks for [application/xml](#). For those reasons — Jackson JSON and Jackson XML will be used as our core marshalling frameworks. JSON-B and JAXB will just be used for portability testing.

Chapter 164. JSON Content

JSON is the content type most preferred by Javascript UI frameworks and NoSQL databases. It has quickly overtaken XML as a preferred data exchange format.

Example JSON Document

```
{  
    "id" : 0,  
    "author" : "Hotblack Desiato",  
    "text" : "Parts of the inside of her head screamed at other parts of the inside of  
her head.",  
    "date" : "1981-05-15"  
}
```

Much of the mapping can be accomplished using Java reflection. Provider-specific annotations can be added to address individual issues. Let's take a look at how both Jackson JSON and JSON-B can be used to map our `QuoteDTO` POJO to the above JSON content. The following is a trimmed down copy of the DTO class I showed you earlier. What kind of things do we need to make that mapping?

Review: Example DTO

```
@Data  
public class QuoteDTO {  
    private int id;  
    private String author;  
    private String text;  
    private LocalDate date; ①  
    private String ignored; ②  
}
```

① may need some LocalDate formatting

② may need to mark as excluded

164.1. Jackson JSON

For the simple cases, our DTO classes can be mapped to JSON with minimal effort using [Jackson JSON](#). However, we potentially need to shape our document and can use [Jackson annotations](#) to customize. The following example shows using an annotation to eliminate a property from the JSON document.

Example Pre-Tweaked JSON Payload

```
{  
    "id" : 0,  
    "author" : "Hotblack Desiato",  
    "text" : "Parts of the inside of her  
head screamed at other parts of the  
inside of her head.",  
    "date" : [ 1981, 5, 15], ①  
    "ignored" : "ignored" ②  
}
```

① LocalDate in a non-ISO array format

② unwanted field included

Example QuoteDTO with Jackson Annotation(s)

```
import  
com.fasterxml.jackson.annotation.JsonIgnore;  
...  
public class QuoteDTO {  
    private int id;  
    private String author;  
    private String text;  
    private LocalDate date;  
    @JsonIgnore ①  
    private String ignored;  
}
```

① Jackson `@JsonIgnore` causes the Java property to be ignored when converting to/from JSON



Date/Time Formatting Handled at ObjectMapper/Marshaller Level

The example annotation above only addressed the `ignore` property. We will address date/time formatting at the ObjectMapper/marshaller level below.

164.1.1. Jackson JSON Initialization

Jackson JSON uses an `ObjectMapper` class to go to/from POJO and JSON. We can configure the mapper with options or configure a reusable builder to create mappers with prototype options. Choosing the latter approach will be useful once we move inside the server.

Jackson JSON Imports

```
import com.fasterxml.jackson.databind.ObjectMapper;  
import com.fasterxml.jackson.databind.SerializationFeature;  
import org.springframework.http.converter.json.Jackson2ObjectMapperBuilder;
```

We have the ability to simply create a default ObjectMapper directly.

Simple Jackson JSON Initialization

```
ObjectMapper mapper = new ObjectMapper();
```

However, when using Spring it is useful to use the Spring `Jackson2ObjectMapperBuilder` class to set many of the data marshalling types for us.

Jackson JSON Initialization using Builder

```
import org.springframework.http.converter.json.Jackson2ObjectMapperBuilder;  
...  
ObjectMapper mapper = new Jackson2ObjectMapperBuilder()
```

```

    .featuresToEnable(SerializationFeature.INDENT_OUTPUT) ①
    .featuresToDisable(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS) ②
        //more later
    .createXmlMapper(false) ③
    .build();

```

- ① optional pretty print indentation
- ② option to use ISO-based strings versus binary values and arrays
- ③ same Spring builder creates both XML and JSON ObjectMappers

Use Injection When Inside Container



When inside the container, have the `Jackson2ObjectMapperBuilder` injected (i.e., not locally-instantiated) in order to pick up external and property configurations/customizations.

By default, Jackson will marshal zone-based timestamps as a decimal number (e.g., `-6106031876.123456789`) and generic date/times as an array of values (e.g., `[1776, 7, 4, 8, 2, 4, 123456789]` and `[1966, 1, 9]`). By disabling this serialization feature, Jackson produces ISO-based strings for all types of timestamps and generic date/times (e.g., `1776-07-04T08:02:04.123456789Z` and `2002-02-14`)

The following [example from the class repository](#) shows a builder customizer being registered as a `@Bean` factory to be able to adjust Jackson defaults used by the server. The returned lambda function is called with a builder each time someone injects a `Jackson2ObjectMapper` — provided the Jackson AutoConfiguration has not been overridden.

Example Jackson2ObjectMapperBuilder Custom Configuration

```

/**
 * Execute these customizations first (Highest Precedence) and then the
 * properties second so that properties can override Java configuration.
 */
@Bean
@Order(Ordered.HIGHEST_PRECEDENCE)
public Jackson2ObjectMapperBuilderCustomizer jacksonMapper() {
    return (builder) -> {
        //spring.jackson.serialization.indent-output=true
        .featuresToEnable(SerializationFeature.INDENT_OUTPUT)
        //spring.jackson.serialization.write-dates-as-timestamps=false
        .featuresToDisable(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS)
        //spring.jackson.date-
format=info.ejava.examples.svc.content.quotes.dto.ISODateFormat
        .dateFormat(new ISODateFormat());
    };
}

```

164.1.2. Jackson JSON Marshalling/Unmarshalling

The mapper created from the builder can then be used to marshal the POJO to JSON.

Marshal DTO to JSON using Jackson

```
private ObjectMapper mapper;

public <T> String marshal(T object) throws IOException {
    StringWriter buffer = new StringWriter();
    mapper.writeValue(buffer, object);
    return buffer.toString();
}
```

The mapper can just as easily—unmarshal the JSON to a POJO instance.

Unmarshal DTO from JSON using Jackson

```
public <T> T unmarshal(Class<T> type, String buffer) throws IOException {
    T result = mapper.readValue(buffer, type);
    return result;
}
```

A packaged set of marshal/unmarshal convenience routines have been packaged inside [ejava-dto-util](#).

164.1.3. Jackson JSON Maven Aspects

For modules with only DTOs with Jackson annotations, only a direct dependency on jackson-annotations is necessary.

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-annotations</artifactId>
</dependency>
```

Modules that will be marshalling/unmarshalling JSON will need the core libraries that can be conveniently brought in through a dependency on one of the following two starters.

- spring-boot-starter-web
- spring-boot-starter-json

```
org.springframework.boot:spring-boot-starter-web:jar
+- org.springframework.boot:spring-boot-starter-json:jar
|   +- com.fasterxml.jackson.core:jackson-databind:jar
|   |   +- com.fasterxml.jackson.core:jackson-annotations:jar
|   |   \- com.fasterxml.jackson.core:jackson-core
```

```

| +- com.fasterxml.jackson.datatype:jackson-datatype-jdk8:jar ①
| +- com.fasterxml.jackson.datatype:jackson-datatype-jsr310:jar ①
| \- com.fasterxml.jackson.module:jackson-module-parameter-names:jar

```

① defines mapping for java.time types



Jackson has built-in ISO mappings for Date and java.time

Jackson has built-in mappings to ISO for `java.util.Date` and `java.time` data types.

164.2. JSON-B

JSON-B (the standard) and [Yasson](#) (the reference implementation of JSON-B) can pretty much render a JSON view of our simple DTO class right out of the box. Customizations can be applied using [JSON-B annotations](#). In the following example, the `ignore` Java property is being excluded from the JSON output.

Example Pre-Tweaked JSON-B Payload

```
{
  "author": "Reg Nullify",
  "date": "1986-05-20", ①
  "id": 0,
  "ignored": "ignored",
  "text": "In the beginning, the Universe
  was created. This has made a lot of
  people very angry and been widely
  regarded as a bad move."
}
```

① LocalDate looks to already be in an ISO-8601 format

Example QuoteDTO with JSON-B Annotation(s)

```

...
import jakarta.json.bind.annotation
.JsonbTransient;
...

public class QuoteDTO {
    private int id;
    private String author;
    private String text;
    private LocalDate date;
    @JsonbTransient ①
    private String ignored;
}

```

① `@JsonbTransient` used to identify unmapped Java properties

164.2.1. JSON-B Initialization

JSON-B provides all mapping through a `Jsonb` builder object that can be configured up-front with various options.

JSON-B Imports

```

import jakarta.json.bind.Jsonb;
import jakarta.json.bind.JsonbBuilder;
import jakarta.json.bind.JsonbConfig;

```

JSON-B Initialization

```
JsonbConfig config=new JsonbConfig()
    .setProperty(JsonbConfig.FORMATTING, true); ①
Jsonb builder = JsonbBuilder.create(config);
```

① adds pretty-printing features to payload

Jsonb is no longer javax

The Jsonb package has changed from `javax.json.bind` to `jakarta.json.bind`.



```
import javax.json.bind.Jsonb; //legacy
import jakarta.json.bind.Jsonb; //modern
```

164.2.2. JSON-B Marshalling/Unmarshalling

The following two examples show how JSON-B marshals and unmarshals the DTO POJO instances to/from JSON.

Marshall DTO using JSON-B

```
private Jsonb builder;

public <T> String marshal(T object) {
    String buffer = builder.toJson(object);
    return buffer;
}
```

Unmarshal DTO using JSON-B

```
public <T> T unmarshal(Class<T> type, String buffer) {
    T result = (T) builder.fromJson(buffer, type);
    return result;
}
```

164.2.3. JSON-B Maven Aspects

Modules defining only the DTO class require a dependency on the following API definition for the annotations.

```
<dependency>
    <groupId>jakarta.json</groupId>
    <artifactId>jakarta.json-api</artifactId>
</dependency>
```

Modules marshalling/unmarshalling JSON documents using JSON-B/Yasson implementation require

dependencies on `binding-api` and a runtime dependency on `yasson` implementation.

```
org.eclipse:yasson:jar
+- jakarta.json.bind:jakarta.json.bind-api:jar
+- jakarta.json:jakarta.json-api:jar
\- org.glassfish:jakarta.json:jar
```

Chapter 165. XML Content

XML is preferred by many data exchange services that require rigor in their data definitions. That does not mean that rigor is always required. The following two examples are XML renderings of a `QuoteDTO`.

The first example is a straight mapping of Java class/attribute to XML elements. The second example applies an XML namespace and attribute (for the `id` property). Namespaces become important when mixing similar data types from different sources. XML attributes are commonly used to host identity information. XML elements are commonly used for description information. The sometimes arbitrary use of attributes over elements in XML leads to some confusion when trying to perform direct mappings between JSON and XML—since JSON has no concept of an attribute.

Example Vanilla XML Document

```
<QuoteDTO> ①
  <id>0</id> ②
  <author>Zaphod Beeblebrox</author>
  <text>Nothing travels faster than the speed of light with the possible exception of
bad news, which obeys its own special laws.</text>
  <date>1927</date> ③
  <date>6</date>
  <date>11</date>
  <ignored>ignored</ignored> ④
</QuoteDTO>
```

① root element name defaults to variant of class name

② all properties default to `@XmlElement` mapping

③ `java.time` types are going to need some work

④ all properties are assumed to not be ignored

Collections Marshall Unwrapped



The three (3) `date` elements above are elements of an ordered collection marshalled without a wrapping element. If we wanted to keep the collection (versus marshalling in ISO format), it would be common to define a wrapping element to encapsulate the collection—much like parentheses in a sentence.

Example XML Document with Namespaces, Attributes, and Desired Shaping

```
<quote xmlns="urn:ejava.svc-controllers.quotes" id="0"> ① ② ③
  <author>Zaphod Beeblebrox</author>
  <text>Nothing travels faster than the speed of light with the possible exception of
bad news, which obeys its own special laws.</text>
  <date>1927-06-11</date>
</quote> ④
```

- ① `quote` is our targeted root element name
- ② `urn:ejava.svc-controllers.quotes` is our targeted namespace
- ③ we want the `id` mapped as an attribute — not an element
- ④ we want certain properties from the DTO not to show up in the XML

165.1. Jackson XML

Like Jackson JSON, [Jackson XML](#) will attempt to map a Java class solely on Java reflection and default mappings. However, to leverage key XML features like namespaces and attributes, we need to add a few [annotations](#). The partial example below shows our POJO with Lombok and other mappings excluded for simplicity.

Example QuoteDTO with Jackson XML Annotations

```
import com.fasterxml.jackson.dataformat.xml.annotation.JacksonXmlProperty;
import com.fasterxml.jackson.dataformat.xml.annotation.JacksonXmlRootElement;
...
@JacksonXmlRootElement(localName = "quote", ①
    namespace = "urn:ejava.svc-controllers.quotes") ②
public class QuoteDTO {
    @JacksonXmlProperty(isAttribute = true) ③
    private int id;
    private String author;
    private String text;
    private LocalDate date;
    @JsonIgnore ④
    private String ignored;
}
```

- ① defines the element name when rendered as the root element
- ② defines namespace for type
- ③ maps `id` property to an XML attribute — default is XML element
- ④ reuses Jackson JSON general purpose annotations

165.1.1. Jackson XML Initialization

Jackson XML initialization is nearly identical to its JSON sibling as long as we want them to have the same options. In all of our examples I will be turning off array-based, numeric dates expression in favor of ISO-based strings.

Jackson XML Imports

```
import com.fasterxml.jackson.databind.SerializationFeature;
import com.fasterxml.jackson.dataformat.xml.XmlMapper;
import org.springframework.http.converter.json.Jackson2ObjectMapperBuilder;
```

Jackson XML Initialization

```
XmlMapper mapper = new Jackson2ObjectMapperBuilder()
    .featuresToEnable(SerializationFeature.INDENT_OUTPUT) ①
    .featuresToDisable(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS) ②
    //more later
    .createXmlMapper(true) ③
    .build();
```

① pretty print output

② use ISO-based strings for time-based fields versus binary numbers and arrays

③ XmlMapper extends ObjectMapper

165.1.2. Jackson XML Marshalling/Unmarshalling

Marshall DTO using Jackson XML

```
public <T> String marshal(T object) throws IOException {
    StringWriter buffer = new StringWriter();
    mapper.writeValue(buffer, object);
    return buffer.toString();
}
```

Unmarshal DTO using Jackson XML

```
public <T> T unmarshal(Class<T> type, String buffer) throws IOException {
    T result = mapper.readValue(buffer, type);
    return result;
}
```

165.1.3. Jackson XML Maven Aspects

Jackson XML is not broken out into separate libraries as much as its JSON sibling. [Jackson XML annotations](#) are housed in the same library as the marshalling/unmarshalling code.

Jackson Dependency for XML-specific Annotation and Marshalling Support

```
<dependency>
    <groupId>com.fasterxml.jackson.dataformat</groupId>
    <artifactId>jackson-dataformat-xml</artifactId>
</dependency>
```

165.2. JAXB

JAXB is more particular about the definition of the Java class to be mapped. JAXB requires that the root element of a document be defined with an `@XmlRootElement` annotation with an optional name and namespace defined.

JAXB Requires `@XmlRootElement` on Root Element of Document

```
com.sun.istack.SAXException2: unable to marshal type  
"info.ejava.examples.svc.content.quotes.dto.QuoteDTO"  
as an element because it is missing an @XmlRootElement annotation
```

Required `@XmlRootElement` supplied

```
...  
import jakarta.xml.bind.annotation.XmlRootElement;  
...  
@XmlRootElement(name = "quote", namespace = "urn:ejava.svc-controllers.quotes")  
public class QuoteDTO { ① ②
```

① default name is `quoteDTO` if not supplied

② default to no namespace if not supplied

JAXB 4.x is no longer javax

JAXB 4.x used in Spring Boot 3/Spring 6 is no longer `javax`. The Java package changed from `javax.xml.bind` to `jakarta.xml.bind`



```
import javax.xml.bind.annotation.XmlRootElement; //Spring Boot 2  
import jakarta.xml.bind.annotation.XmlRootElement; //Spring Boot 3
```

165.2.1. Custom Type Adapters

JAXB has no default definitions for `java.time` classes and must be handled with custom adapter code.

JAXB has no default mapping for java.time classes

```
INFO: No default constructor found on class java.time.LocalDate  
java.lang.NoSuchMethodException: java.time.LocalDate.<init>()
```

This has always been an issue for Date formatting even before `java.time` and can easily be solved with a custom adapter class that converts between a String and the unsupported type. We can locate [packaged solutions](#) on the web, but it is helpful to get comfortable with the process on our own.

We first create an adapter class that extends `XmlAdapter<ValueType, BoundType>`—where `ValueType` is a type known to JAXB and `BoundType` is the type we are mapping. We can use `DateFormatter.ISO_LOCAL_DATE` to marshal and unmarshal the `LocalDate` to/from text.

Example JAXB LocalDate Adapter

```
import jakarta.xml.bind.annotation.adapters.XmlAdapter;  
...
```

```

public static class LocalDateJaxbAdapter extends XmlAdapter<String, LocalDate>
{
    @Override
    public LocalDate unmarshal(String text) {
        return text == null ? null : LocalDate.parse(text, DateTimeFormatter.ISO_LOCAL_DATE);
    }
    @Override
    public String marshal(LocalDate timestamp) {
        return timestamp==null ? null : DateTimeFormatter.ISO_LOCAL_DATE.format(timestamp);
    }
}

```

We next annotate the Java property with `@XmlJavaTypeAdapter`, naming our adapter class.

Example Mapping Custom Type to Adapter for Class Property

```

import jakarta.xml.bind.annotation.adapters.XmlJavaTypeAdapter;
...
@XmlAccessorType(XmlAccessType.FIELD) ②
public class QuoteDTO {
...
    @XmlJavaTypeAdapter(LocalDateJaxbAdapter.class) ①
    private LocalDate date;

```

① custom adapter required for unsupported types

② must manually set access to FIELD when annotating attributes

165.2.2. JAXB Initialization

There is no sharable, up-front initialization for JAXB. All configuration must be done on individual, non-sharable JAXBContext objects. However, JAXB does have a package-wide annotation that the other frameworks do not. The following example shows a `package-info.java` file that contains annotations to be applied to every class in the same Java package.

JAXB Package Annotations

```

//package-info.java
@XmlSchema(namespace = "urn:ejava.svc-controllers.quotes")
package info.ejava.examples.svc.content.quotes.dto;

import jakarta.xml.bind.annotation.XmlSchema;

```

The same feature could be used to globally apply adapters package-wide.

Example Mapping Custom Type to Adapter for Package

```
//package-info.java
```

```

@XmlSchema(namespace = "urn:ejava.svc-controllers.quotes")
@XmlJavaTypeAdapter(type= LocalDate.class, value=JaxbTimeAdapters.
LocalDateJaxbAdapter.class)
package info.ejava.examples.svc.content.quotes.dto;

import jakarta.xml.bind.annotation.XmlSchema;
import jakarta.xml.bind.annotation.adapters.XmlJavaTypeAdapter;
import java.time.LocalDate;

```

165.2.3. JAXB Marshalling/Unmarshalling

JAXB Imports

```

import jakarta.xml.bind.JAXBContext;
import jakarta.xml.bind.JAXBException;
import jakarta.xml.bind.Marshaller;
import jakarta.xml.bind.Unmarshaller;

```

Marshalling/Unmarshalling starts out by constructing a **JAXBContext** scoped to handle the classes of interest. This will include the classes explicitly named and the classes they reference. Therefore, one would only need to create a **JAXBContext** by explicitly naming the input and return types of a Web API method.

Marshall DTO using JAXB

```

public <T> String marshal(T object) throws JAXBException {
    JAXBContext jbx = JAXBContext.newInstance(object.getClass()); ①
    Marshaller marshaller = jbx.createMarshaller();
    marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true); ②

    StringWriter buffer = new StringWriter();
    marshaller.marshal(object, buffer);
    return buffer.toString();
}

```

① explicitly name primary classes of interest

② adds newline and indentation formatting

Unmarshal DTO using JAXB

```

public <T> T unmarshal(Class<T> type, String buffer) throws JAXBException {
    JAXBContext jbx = JAXBContext.newInstance(type);
    Unmarshaller unmarshaller = jbx.createUnmarshaller();

    ByteArrayInputStream bis = new ByteArrayInputStream(buffer.getBytes());
    T result = (T) unmarshaller.unmarshal(bis);
    return result;
}

```

165.2.4. JAXB Maven Aspects

Modules that define DTO classes only will require a direct dependency on the `jakarta.xml-bind-api` library for annotations and interfaces.

```
<dependency>
    <groupId>jakarta.xml.bind</groupId>
    <artifactId>jakarta.xml.bind-api</artifactId>
</dependency>
```

Modules marshalling/unmarshalling DTO classes using JAXB will require a dependency on the `jAXB-runtime` artifact.

```
<dependency>
    <groupId>org.glassfish.jaxb</groupId>
    <artifactId>jaxb-runtime</artifactId>
</dependency>
```

Deprecated javax.xml.bind Dependencies

The `jAXB-api` artifact contains the deprecated `javax.xml.bind` interface types.

```
<dependency>
    <groupId>javax.xml.bind</groupId>
    <artifactId>jaxb-api</artifactId>
</dependency>
```

The following two artifacts contain the deprecated `javax.xml.bind` implementation classes. `jAXB-core` contains visible utilities used map between Java and XML Schema. `jAXB-impl` is more geared towards runtime. Since both are needed, I am not sure why there is not a dependency between one another to make that automatic.



```
<dependency>
    <groupId>com.sun.xml.bind</groupId>
    <artifactId>jaxb-core</artifactId>
</dependency>
<dependency>
    <groupId>com.sun.xml.bind</groupId>
    <artifactId>jaxb-impl</artifactId>
</dependency>
```

Chapter 166. Configure Server-side Jackson

166.1. Dependencies

Jackson JSON will already be on the classpath when using `spring-boot-web-starter`. To also support XML, make sure the server has an additional `jackson-dataformat-xml` dependency.

Server-side Dependency Required for Jackson XML Support

```
<dependency>
    <groupId>com.fasterxml.jackson.dataformat</groupId>
    <artifactId>jackson-dataformat-xml</artifactId>
</dependency>
```

166.2. Configure ObjectMapper

Both XML and JSON mappers are instances of `ObjectMapper`. To configure their use in our application—we can go one step higher and create a builder for jackson to use as its base. That is all we need to know as long as we can configure them identically.

Jackson's AutoConfiguration provides a layered approach to customizing the marshaller. One can configure using:

- `spring.jackson properties` (e.g., `spring.jackson.serialization.*`)
- `Jackson2ObjectMapperBuilderCustomizer`—a functional interface that will be passed a builder pre-configured using properties

Assigning a high precedence order to the customizer will allow properties to flexibly override the Java code configuration.

Server-side Jackson Builder @Bean Factory

```
...
import com.fasterxml.jackson.databind.SerializationFeature;
import
org.springframework.boot.autoconfigure.jackson.Jackson2ObjectMapperBuilderCustomizer;
import org.springframework.core.Ordered;
import org.springframework.core.annotation.Order;

@SpringBootApplication
public class QuotesApplication {
    public static void main(String...args) {
        SpringApplication.run(QuotesApplication.class, args);
    }

    @Bean
    @Order(Ordered.HIGHEST_PRECEDENCE) ②
    public Jackson2ObjectMapperBuilderCustomizer jacksonMapper() {
```

```

    return (builder) -> { builder ①
        //spring.jackson.serialization.indent-output=true
        .featuresToEnable(SerializationFeature.INDENT_OUTPUT)
        //spring.jackson.serialization.write-dates-as-timestamps=false
        .featuresToDisable(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS)
        //spring.jackson.date-
format=info.ejava.examples.svc.content.quotes.dto.ISODateFormat
        .dateFormat(new ISODateFormat());
    };
}

```

- ① returns a lambda function that is called with a `Jackson2ObjectMapperBuilder` to customize. Jackson uses this same definition for both XML and JSON mappers
- ② highest order precedence applies this configuration first, then properties—allowing for overrides using properties

166.3. Controller Properties

We can register what MediaTypes each method supports by adding a set of consumes and produces properties to the `@RequestMapping` annotation in the controller. This is an array of MediaType values (e.g., `["application/json", "application/xml"]`) that the endpoint should either accept or provide in a response.

Example Consumes and Produces Mapping

```

@RequestMapping(path= QUOTES_PATH,
    method= RequestMethod.POST,
    consumes = {MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE
},
    produces = {MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE
})
public ResponseEntity<QuoteDTO> createQuote(@RequestBody QuoteDTO quote) {
    QuoteDTO result = quotesService.createQuote(quote);

    URI uri = ServletUriComponentsBuilder.fromCurrentRequestUri()
        .replacePath(QUOTE_PATH)
        .build(result.getId());
    ResponseEntity<QuoteDTO> response = ResponseEntity.created(uri)
        .body(result);
    return response;
}

```

The `Content-Type` request header is matched with one of the types listed in `consumes`. This is a single value and the following example uses an `application/json` Content-Type and the server uses our Jackson JSON configuration and DTO mappings to turn the JSON string into a POJO.

Example POST of JSON Content

```
POST http://localhost:64702/api/quotes
```

```

sent: [Accept:"application/xml", Content-Type:"application/json", Content-Length:"108"]
[
{
  "id" : 0,
  "author" : "Tricia McMillan",
  "text" : "Earth: Mostly Harmless",
  "date" : "1991-05-11"
}

```

If there is a match between Content-Type and consumes, the provider will map the body contents to the input type using the mappings we reviewed earlier. If we need more insight into the request headers—we can change the method mapping to accept a RequestEntity and obtain the headers from that object.

Example Alternative Content Mapping

```

@RequestMapping(path= QUOTES_PATH,
    method= RequestMethod.POST,
    consumes={MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE},
    produces={MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE})
//  public ResponseEntity<QuoteDTO> createQuote(@RequestBody QuoteDTO quote) {
public ResponseEntity<QuoteDTO> createQuote(RequestEntity<QuoteDTO> request) {❶
    QuoteDTO quote = request.getBody();
    log.info("CONTENT_TYPE={}", request.getHeaders().get(HttpHeaders.CONTENT_TYPE));
    log.info("ACCEPT={}", request.getHeaders().get(HttpHeaders.ACCEPT));
    QuoteDTO result = quotesService.createQuote(quote);
}

```

❶ injecting raw input RequestEntity versus input payload to inspect header properties

The log statements at the start of the methods output the following two lines with request header information.

Example Header Output

```

QuotesController#createQuote:38 CONTENT_TYPE=[application/json;charset=UTF-8]
QuotesController#createQuote:39 ACCEPT=[application/xml]

```

Whatever the service returns (success or error), the `Accept` request header is matched with one of the types listed in the `produces`. This is a list of N values listed in priority order. In the following example, the client used an `application/xml` Accept header and the server converted it to XML using our Jackson XML configuration and mappings to turn the POJO into an XML response.

Review: Original Request Headers

```

sent: [Accept:"application/xml", Content-Type:"application/json", Content-
Length:"108"]

```

Response Header and Payload

```
rcvd: [Location:"http://localhost:64702/api/quotes/1", Content-Type:"application/xml",
Transfer-Encoding:"chunked", Date:"Fri, 05 Jun 2020 19:44:25 GMT", Keep-
Alive:"timeout=60", Connection:"keep-alive"]
<quote xmlns="urn:ejava.svc-controllers.quotes" id="1">
  <author xmlns="">Tricia McMillan</author>
  <text xmlns="">Earth: Mostly Harmless</text>
  <date xmlns="">1991-05-11</date>
</quote>
```

If there is no match between Content-Type and consumes, a [415/Unsupported Media Type](#) error status is returned. If there is no match between Accept and produces, a [406/Not Acceptable](#) error status is returned. Most of this content negotiation and data marshalling/unmarshalling is hidden from the controller.

Chapter 167. Client Marshall Request Content

If we care about the exact format our POJO is marshalled to or the format the service returns, we can no longer pass a naked POJO to the client library. We must wrap the POJO in a [RequestEntity](#) and supply a set of headers with format specifications. The following shows an example using [RestTemplate](#).

RestTemplate Content Headers Example

```
RequestEntity<QuoteDTO> request = RequestEntity.post(quotesUrl) ①
    .contentType(contentType) ②
    .accept(acceptType) ③
    .body(validQuote);
ResponseEntity<QuoteDTO> response = restTemplate.exchange(request, QuoteDTO.class);
```

① create a POST request with client headers

② express desired Content-Type for the request

③ express Accept types for the response

The following example shows the request and reply information exchange for an [application/json](#) Content-Type and Accept header.

Example JSON POST Request and Reply

```
POST http://localhost:49252/api/quotes, returned CREATED/201
sent: [Accept:"application/json", Content-Type:"application/json", Content-Length:"146"]
{
  "id" : 0,
  "author" : "Zarquon",
  "text" : "Whatever your tastes, Magrathea can cater for you. We are not proud.",
  "date" : "1920-08-17"
}
rcvd: [Location:"http://localhost:49252/api/quotes/1", Content-Type:"application/json",
", Transfer-Encoding:"chunked", Date:"Fri, 05 Jun 2020 20:17:35 GMT", Keep-Alive:"timeout=60", Connection:"keep-alive"]
{
  "id" : 1,
  "author" : "Zarquon",
  "text" : "Whatever your tastes, Magrathea can cater for you. We are not proud.",
  "date" : "1920-08-17"
}
```

The following example shows the request and reply information exchange for an [application/xml](#) Content-Type and Accept header.

Example XML POST Request and Reply

```
POST http://localhost:49252/api/quotes, returned CREATED/201
sent: [Accept:"application/xml", Content-Type:"application/xml", Content-Length:"290"]
<quote xmlns="urn:ejava.svc-controllers.quotes" id="0">
    <author xmlns="">Humma Kavula</author>
    <text xmlns="">In the beginning, the Universe was created. This has made a lot of
people very angry and been widely regarded as a bad move.</text>
    <date xmlns="">1942-03-03</date>
</quote>

rcvd: [Location:"http://localhost:49252/api/quotes/4", Content-Type:"application/xml",
Transfer-Encoding:"chunked", Date:"Fri, 05 Jun 2020 20:17:35 GMT", Keep-
Alive:"timeout=60", Connection:"keep-alive"]
<quote xmlns="urn:ejava.svc-controllers.quotes" id="4">
    <author xmlns="">Humma Kavula</author>
    <text xmlns="">In the beginning, the Universe was created. This has made a lot of
people very angry and been widely regarded as a bad move.</text>
    <date xmlns="">1942-03-03</date>
</quote>
```

Chapter 168. Client Filters

The runtime examples above showed HTTP traffic and marshalled payloads. That can be very convenient for debugging purposes. There are two primary ways of examining marshalled payloads.

Switch accepted Java type to String

Both our client and controller declare they expect a `QuoteDTO.class` to be the response. That causes the provider to map the String into the desired type. If the client or controller declared they expected a `String.class`, they would receive the raw payload to debug or later manually parse using direct access to the unmarshalling code.

Add a filter

Both `RestTemplate` and `WebClient` accept filters in the request and response flow. `RestTemplate` is easier and more capable to use because of its synchronous behavior. We can register a filter to get called with the full request and response in plain view—with access to the body—using `RestTemplate`. `WebClient`, with its asynchronous design has a separate request and response flow with no easy access to the payload.

168.1. RestTemplate and RestClient

The following [code provides an example of a filter](#) that will work for the synchronous `RestTemplate` and `RestClient`. It shows the steps taken to access the request and response payload. Note that reading the body of a request or response is commonly a read-once restriction. The ability to read the body multiple times will be taken care of within the `@Bean` factory method registering this filter.

Example RestTemplate/RestClient Logging Filter

```
import org.springframework.http.client.ClientHttpRequestExecution;
import org.springframework.http.client.ClientHttpRequestInterceptor;
import org.springframework.http.client.ClientHttpResponse;

...
public class RestTemplateLoggingFilter implements ClientHttpRequestInterceptor {
    public ClientHttpResponse intercept(HttpRequest request, byte[] body,①
                                         ClientHttpRequestExecution execution) throws IOException {
        ClientHttpResponse response = execution.execute(request, body); ①
        HttpMethod method = request.getMethod();
        URI uri = request.getURI();
        HttpStatusCode status = response.getStatusCode();
        String requestBody = new String(body);
        String responseBody = this.readString(response.getBody());
        //... log debug
        return response;
    }
    private String readString(InputStream inputStream) { ... }
    ...
}
```

① interceptor has access to the client request and response

RestTemplateLoggingFilter is for all Synchronous Requests



The example class is called `RestTemplateLoggingFilter` because `RestTemplate` was here first, the filter is used many times in many examples, and I did not want to make the generalized name change at this time. It is specific to synchronous requests, which includes `RestClient`.

The following code shows an example of a `@Bean` factory that creates `RestTemplate` instances configured with the debug logging filter shown above.

Example @Bean Factory Registering RestTemplate Filter

```
@Bean
ClientHttpRequestFactory requestFactory() {
    return new SimpleClientHttpRequestFactory(); ③
}

@Bean
public RestTemplate restTemplate(RestTemplateBuilder builder,
    ClientHttpRequestFactory requestFactory) { ③
    return builder.requestFactory(
        //used to read the streams twice -- so we can use the logging filter
        ()->new BufferingClientHttpRequestFactory(requestFactory)) ②
        .interceptors(new RestTemplateLoggingFilter()) ①
        .build();
}

@Bean
public RestClient restClient(RestClient.Builder builder,
    ClientHttpRequestFactory requestFactory) { ③
    return builder //requestFactory used to read stream twice
        .requestFactory(new BufferingClientHttpRequestFactory(requestFactory)) ②
        .requestInterceptor(new RestTemplateLoggingFilter()) ①
        .build();
}
```

① the overall intent of this `@Bean` factory is to register the logging filter

② must configure client with a buffer (`BufferingClientHttpRequestFactory`) for body to enable multiple reads

③ providing a `ClientRequestFactory` to be forward-ready for SSL communications

168.2. WebClient

The following code shows an example request and response filter. They are independent and are implemented using a Java 8 lambda function. You will notice that we have no easy access to the request or response body.

Example WebClient Logging Filter

```
package info.ejava.examples.common.webflux;

import org.springframework.web.reactive.function.client.ExchangeFilterFunction;
...
public class WebClientLoggingFilter {
    public static ExchangeFilterFunction requestFilter() {
        return ExchangeFilterFunction.ofRequestProcessor((request) -> {
            //access to
            //request.method(),
            //request.url(),
            //request.headers()
            return Mono.just(request);
        });
    }
    public static ExchangeFilterFunction responseFilter() {
        return ExchangeFilterFunction.ofResponseProcessor((response) -> {
            //access to
            //response.statusCode()
            //response.headers().asHttpHeaders()
            return Mono.just(response);
        });
    }
}
```

The code below demonstrates how to register custom filters for injected WebClient instances.

Example @Bean Factory Registering WebClient Filters

```
@Bean
public WebClient webClient(WebClient.Builder builder) {
    return builder
        .filter(WebClientLoggingFilter.requestFilter())
        .filter(WebClientLoggingFilter.responseFilter())
        .build();
}
```

Chapter 169. Date/Time Lenient Parsing and Formatting

In our quote example, we had an easy LocalDate to format and parse, but that even required a custom adapter for JAXB. Integration of other time-based properties can get more involved as we get into complete timestamps with timezone offsets. So let's try to address the issues here before we complete the topic on content exchange.

The primary time-related issues we can encounter include:

Table 14. Potential Time-related Format Issues

Potential Issue	Description
type not supported	We have already encountered that with JAXB and solved using a custom adapter. Each of the providers offer their own form of adapter (or serializer/deserializer), so we have a good headstart on how to solve the hard problems.
non-UTC ISO offset style supported	There are at least four or more expressions of a timezone offset (Z, +00, +0000, or +00:00) that could be used. Not all of them can be parsed by each provider out-of-the-box.
offset versus extended offset zone formatting	There are more verbose styles (Z[UTC]) of expressing timezone offsets that include the ZoneId
fixed width or truncated	Are all fields supplied at all times even when they are 0 (e.g., 1776-07-04T00:00:00.100+00:00) or are values truncated to only include significant values (e.g., '1776-07-04T00:00:00.1Z'). This mostly applies to fractions of seconds.

We should always strive for:

- consistent (ISO) standard format to marshal time-related fields
- leniently parsing as many formats as possible

Let's take a look at establishing an internal standard, determining which providers violate that standard, how to adjust them to comply with our standard, and how to leniently parse many formats with the Jackson parser since that will be our standard provider for the course.

169.1. Out of the Box Time-related Formatting

Out of the box, I found the providers marshalled `OffsetDateTime` and `Date` with the following format. I provided an `OffsetDateTime` and `Date` timestamp with varying number of nanoseconds (123456789, 1, and 0 ns) and timezone UTC and -05:00 and the following table shows what was marshalled for the DTO.

Table 15. Default Provider OffsetDateTime and Date Formats

Provider	OffsetDateTime	Trunc	Date	Trunc
Jackson	1776-07-04T00:00:00.123456789Z 1776-07-04T00:00:00.1Z 1776-07-04T00:00:00Z 1776-07-03T19:00:00.123456789-05:00 1776-07-03T19:00:00.1-05:00 1776-07-03T19:00:00-05:00	Yes	1776-07-04T00:00:00.123+00:00 1776-07-04T00:00:00.100+00:00 1776-07-04T00:00:00.000+00:00	No
JSON-B	1776-07-04T00:00:00.123456789Z 1776-07-04T00:00:00.1Z 1776-07-04T00:00:00Z 1776-07-03T19:00:00.123456789-05:00 1776-07-03T19:00:00.1-05:00 1776-07-03T19:00:00-05:00	Yes	1776-07-04T00:00:00.123Z[UTC] 1776-07-04T00:00:00.1Z[UTC] 1776-07-04T00:00:00Z[UTC]	Yes
JAXB	(not supported/ custom adapter required)	n/a	1776-07-03T19:00:00.123-05:00 1776-07-03T19:00:00.100-05:00 1776-07-03T19:00:00-05:00	Yes/ No

Jackson and JSON-B—out of the box—use an ISO format that truncates nanoseconds and uses "Z" and "+00:00" offset styles for `java.time` types. JAXB does not support `java.time` types. When a non-UTC time is supplied, the time is expressed using the targeted offset. You will notice that Date is always modified to be UTC.

Jackson Date format is a fixed length, no truncation, always expressed at UTC with an `+HH:MM` expressed offset. JSON-B and JAXB Date formats truncate milliseconds/nanoseconds. JSON-B uses extended timezone offset (`Z[UTC]`) and JAXB uses "+00:00" format. JAXB also always expresses the Date in `EST` in my case.

169.2. Out of the Box Time-related Parsing

To cut down on our choices, I took a look at which providers out-of-the-box could parse the different timezone offsets. To keep things sane, my detailed focus was limited to the Date field. The table shows that each of the providers can parse the "Z" and "+00:00" offset format. They were also able to process variable length formats when faced with less significant nanosecond cases.

Table 16. Default Can Parse Formats

Provider	ISO Z	ISO +00	ISO +0000	ISO +00:00	ISO Z[UTC]
Jackson	Yes	Yes	Yes	Yes	No
JSON-B	Yes	No	No	Yes	Yes
JAXB	Yes	No	No	Yes	No

The testing results show that timezone expressions "Z" or "+00:00" format should be portable and something to target as our marshalling format.

- Jackson - no output change
- JSON-B - requires modification

- JAXB - requires no change

169.3. JSON-B DATE_FORMAT Option

We can configure JSON-B time-related field output using a `java.time` format string. `java.time` permits optional characters. `java.text` does not. The following expression is good enough for Date output but will create a parser that is intolerant of varying length timestamps. For that reason, I will not choose the type of option that locks formatting with parsing.

JSON-B global DATE_FORMAT Option

```
JsonbConfig config=new JsonbConfig()
    .setProperty(JsonbConfig.DATE_FORMAT, "yyyy-MM-dd'T'HH:mm:ss[.SSS][XXX]") ①
    .setProperty(JsonbConfig.FORMATTING, true);
builder = JsonbBuilder.create(config);
```

① a fixed formatting and parsing candidate option rejected because of parsing intolerance

169.4. JSON-B Custom Serializer Option

A better JSON-B solution would be to create a serializer — independent of deserializer — that takes care of the formatting.

Example JSON-B Default Serializer

```
public class DateJsonbSerializer implements JsonbSerializer<Date> {
    @Override
    public void serialize(Date date, JsonGenerator generator, SerializationContext serializationContext) {
        generator.write(DateTimeFormatter.ISO_INSTANT.format(date.toInstant()));
    }
}
```

We add `@JsonbTypeSerializer` annotation to the field we need to customize and supply the class for our custom serializer.

Example JSON-B Annotation Applied

```
@JsonbTypeSerializer(JsonbTimeSerializers.DateJsonbSerializer.class)
private Date date;
```

With the above annotation in place and the `JsonConfig` unmodified, we get output format we want from JSON-B without impacting its built-in ability to parse various time formats.

- 1776-07-04T00:00:00.123Z
- 1776-07-04T00:00:00.100Z
- 1776-07-04T00:00:00Z

169.5. Jackson Lenient Parser

All those modifications shown so far are good, but we would also like to have lenient input parsing—possibly more lenient than built into the providers. Jackson provides the ability to pass in a `SimpleDateFormat` format string or an instance of class that extends `DateFormat`. `SimpleDateFormat` does not make a good lenient parser, therefore I created a lenient parser that uses `DateTimeFormatter` framework and plugged that into the `DateFormat` framework.

Example Custom DateFormat Class Implementing Lenient Parser

```
public class ISODateFormat extends DateFormat implements Cloneable {
    public static final DateTimeFormatter UNMARSHALLER = new DateTimeFormatterBuilder()
() {
    //...
        .toFormatter();
    public static final DateTimeFormatter MARSHALLER = DateTimeFormatter
.ISO_OFFSET_DATE_TIME;
    public static final String MARSHAL_ISO_DATE_FORMAT = "yyyy-MM-
dd'T'HH:mm:ss[.SSS]XXX";
}

@Override
public Date parse(String source, ParsePosition pos) {
    OffsetDateTime odt = OffsetDateTime.parse(source, UNMARSHALLER);
    pos.setIndex(source.length()-1);
    return Date.from(odt.toInstant());
}
@Override
public StringBuffer format(Date date, StringBuffer toAppendTo, FieldPosition pos)
{
    ZonedDateTime zdt = ZonedDateTime.ofInstant(date.toInstant(), ZoneOffset.UTC);
    MARSHALLER.formatTo(zdt, toAppendTo);
    return toAppendTo;
}
@Override
public Object clone() {
    return new ISODateFormat(); //we have no state to clone
}
}
```

I have built the lenient parser using the Java interface to `DateTimeFormatter`. It is designed to

- handle variable length time values
- different timezone offsets
- a few different timezone offset expressions

DateTimeFormatter Lenient Parser Definition

```
public static final DateTimeFormatter UNMARSHALLER = new DateTimeFormatterBuilder()
.parseCaseInsensitive()
```

```

.append(DateTimeFormatter.ISO_LOCAL_DATE)
.appendLiteral('T')
.append(DateTimeFormatter.ISO_LOCAL_TIME)
.parseLenient()
.optionalStart().appendOffset("+HH", "Z").optionalEnd()
.optionalStart().appendOffset("+HH:mm", "Z").optionalEnd()
.optionalStart().appendOffset("+HHmm", "Z").optionalEnd()
.optionalStart().appendLiteral('[').parseCaseSensitive()
    .appendZoneRegionId()
    .appendLiteral(']').optionalEnd()
.parseDefaulting(ChronoField.OFFSET_SECONDS, 0)
.parseStrict()
.toFormatter();

```

An instance of my `ISODateFormat` class is then registered with the provider to use on all interfaces.

```

mapper = new Jackson2ObjectMapperBuilder()
    .featuresToEnable(SerializationFeature.INDENT_OUTPUT)
    .featuresToDisable(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS)
    .dateFormat(new ISODateFormat()) ①
    .createXmlMapper(false)
    .build();

```

① registering a global time formatter for Dates

In the server, we can add that same configuration option to our builder `@Bean` factory.

```

@Bean
public Jackson2ObjectMapperBuilderCustomizer jacksonMapper() {
    return (builder) -> { builder
        .featuresToEnable(SerializationFeature.INDENT_OUTPUT)
        .featuresToDisable(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS)
        .dateFormat(new ISODateFormat()); ①
    };
}

```

① registering a global time formatter for Dates for JSON and XML

At this point we have the insights into time-related issues and knowledge of how we can correct.

Chapter 170. Summary

In this module we:

- introduces the DTO pattern and contrasted it with the role of the Business Object
- implemented a DTO class with several different types of fields
- mapped our DTOs to/from a JSON and XML document using multiple providers
- configured data mapping providers within our server
- identified integration issues with time-related fields and learned how to create custom adapters to help resolve issues
- learned how to implement client filters
- took a deeper dive into time-related formatting issues in content and ways to address

Swagger

copyright © 2024 jim stafford (jim.stafford@jhu.edu)

Chapter 171. Introduction

The core charter of this course is to introduce you to framework solutions in Java and focus on core Spring and SpringBoot frameworks. **Details** of Web APIs, database design, and distributed application design are best covered in other courses. We have been covering a modest amount of Web API topics in these last set of modules to provide a functional front door to our application implementations. You know by now how to implement basic RMM level 2, CRUD Web APIs. I now want to wrap up the Web API coverage by introducing a functional way to call those Web APIs with minimal work using Swagger UI. Detailed aspects of configuring Swagger UI is considered out of scope for this course but many example implementation details are included in each API example and a detailed example in the [Swagger Contest Example](#).

171.1. Goals

You will learn to:

- identify the items in the Swagger landscape and its central point — OpenAPI
- generate an Open API interface specification from Java code
- deploy and automatically configure a Swagger UI based on your Open API interface specification
- invoke Web API endpoint operations using Swagger UI
- generate a client library

171.2. Objectives

At the conclusion of this lecture and related exercises, you will be able to:

1. generate a default Open API 3.0 interface specification using Springdoc
2. configure and deploy a Swagger UI that calls your Web API using the Open API specification generated by your API
3. make HTTP CRUD interface calls to your Web API using Swagger UI
4. identify the starting point to make configuration changes to Springdoc
5. generate a client API using the OpenAPI schema definition from a running application

Chapter 172. Swagger Landscape

The core portion of the [Swagger](#) landscape is made up of a line of standards and products geared towards HTTP-based APIs and supported by the company [SmartBear](#). There are two types of things directly related to Swagger: the OpenAPI standard and tools. Although heavily focused on Java implementations, Swagger is generic to all HTTP API providers and not specific to Spring.

172.1. Open API Standard

[OpenAPI](#)—is an implementation-agnostic interface specification for HTTP-based APIs. This was originally baked into the Swagger tooling but donated to open source community in 2015 as a way to define and document interfaces.

- [Open API 2.0](#) - released in 2014 as the last version prior to transitioning to open source. This is equivalent to the Swagger 2.0 Specification.
- [Open API 3.x](#) - released in 2017 as the first version after transitioning to open source.

172.2. Swagger-based Tools

Within the close Swagger umbrella, there are a set of [Tools](#), both free/open source and commercial that are largely provided by Smartbear.

- Swagger Open Source Tools - these tools are primarily geared towards single API at a time uses.
 - [Swagger UI](#)—is a user interface that can be deployed remotely or within an application. This tool displays descriptive information and provides the ability to execute API methods based on a provided OpenAPI specification.
 - Swagger Editor - is a tool that can be used to create or augment an OpenAPI specification.
 - Swagger Codegen - is a tool that builds server stubs and client libraries for APIs defined using OpenAPI.
- Swagger Commercial Tools - these tools are primarily geared towards enterprise usage.
 - Swagger Inspector - a tool to create OpenAPI specifications using external call examples
 - Swagger Hub - repository of OpenAPI definitions

SmartBear offers another set of open source and commercial test tools called [SoapUI](#) which is geared at authoring and executing test cases against APIs and can read in OpenAPI as one of its API definition sources.

Our only requirement in going down this Swagger path is to have the capability to invoke HTTP methods of our endpoints with some ease. There are at least two libraries that focus on generating the Open API spec and packaging a version of the Swagger UI to document and invoke the API in Spring Boot applications: Springfox and Springdocs.

172.3. Springfox

[Springfox](#) is focused on delivering Swagger-based solutions to Spring-based API implementations but is not an official part of Spring, Spring Boot, or Smartbear. It is hard to even find a link to Springfox on the Spring documentation web pages.

Essentially Springfox is:

- a means to generate Open API specs using Java annotations
- a packaging and light configuring of the Swagger-provided swagger UI

Springfox has been around many years. I found the [initial commit](#) in 2012. It supported Open API 2.0 when I originally looked at it in June 2020 (Open API 3.0 was released in 2017). At that time, the Webflux branch was also still in SNAPSHOT. However, a few weeks later a flurry of [releases](#) went out that included Webflux support but no releases have occurred in the years since. Consider it deceased relative to Spring Boot 3.

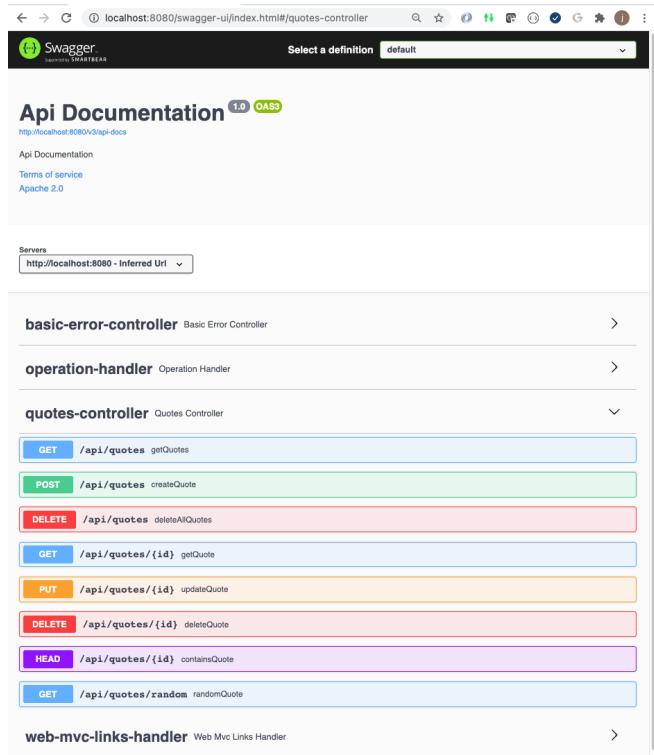


Figure 57. Example Springfox Swagger UI

Springfox does not work with >= Spring Boot 2.6



Springfox does not work with Spring Boot >= 2.6 where a patternParser was deprecated and causes an inspection error during initialization. We can work around the issue for demonstration — but serious use of Swagger (as of July 2022) is now limited to Springdoc.

Springfox is dead



There has not been a single git commit to Springfox since June 2020 and that version does not work with Spring Boot 3. We will consider Springfox dead and not speak much of it further.

172.4. Springdoc

[Springdoc](#) is an independent project focused on delivering Swagger-based solutions to Spring Boot APIs. Like Springfox, Springdoc has no official ties to Spring, Spring Boot, or Pivotal Software. The library was created because of Springfox's lack of support for Open API 3.x many years after its release.

Springdoc is relatively new compared to Springfox. I found its [initial commit](#) in July 2019 and has released several [versions](#) per month since—until 2023. That indicates to me that they had a lot of catch-up to do to complete the product and now are in a relative maintenance mode. They had the advantage of coming in when the standard was more mature and were able to bypass earlier Open API versions. Springdoc targets integration with the latest Spring Web API frameworks—including Spring MVC and Spring WebFlux.

The 1.x version of the library is compatible with Spring Boot 2.x. The 2.x version of the library has been updated to use jakarta dependencies and is required for Spring Boot 3.x.

The screenshot shows the Springdoc SwaggerUI interface at the URL <http://localhost:8080/swagger-ui/index.html?configUrl=v3/api-docs/swagger-config>. The top navigation bar includes links for 'OpenAPI definition' (with a 'GATEWAY' badge), 'Explore', and 'Logout'. Below the navigation, there's a 'Servers' dropdown set to 'http://localhost:8080 - Generated server url'. The main content area is titled 'quotes-controller'. It lists various API endpoints with their HTTP methods and URLs: GET /api/quotes, POST /api/quotes, DELETE /api/quotes, GET /api/quotes/{id}, PUT /api/quotes/{id}, DELETE /api/quotes/{id}, HEAD /api/quotes/{id}, and GET /api/quotes/random. Each endpoint is color-coded (blue, green, red, blue, orange, red, purple, blue). Below the endpoints, there's a 'Schemas' section with two items: 'QuoteDTO' and 'QuoteListDTO', each with a small arrow icon.

Figure 58. Example Springdoc SwaggerUI

Chapter 173. Minimal Configuration

My goal in bringing the Swagger topics into the course is solely to provide us with a convenient way to issue example calls to our API—which is driving our technology solution within the application. For that reason, I am going to show the least amount of setup required to enable a Swagger UI and have it do the default thing.

The minimal configuration will be missing descriptions for endpoint operations, parameters, models, and model properties. The content will rely solely on interpreting the Java classes of the controller, model/DTO classes referenced by the controller, and their annotations. Springdoc definitely does a better job at figuring out things automatically, but they are both functional in this state.

173.1. Springdoc Minimal Configuration

Springdoc minimal configuration is as simple as it gets. All that is required is a single Maven dependency.

173.1.1. Springdoc Maven Dependency

Springdoc has a single top-level dependency that brings in many lower-level dependencies. This specific artifact changed between Spring Boot 2 and 3

Springdoc Spring Boot 2 Maven Dependency

```
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-ui</artifactId>
</dependency>
```

Springdoc Spring Boot 3 Maven Dependency

```
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
</dependency>
```

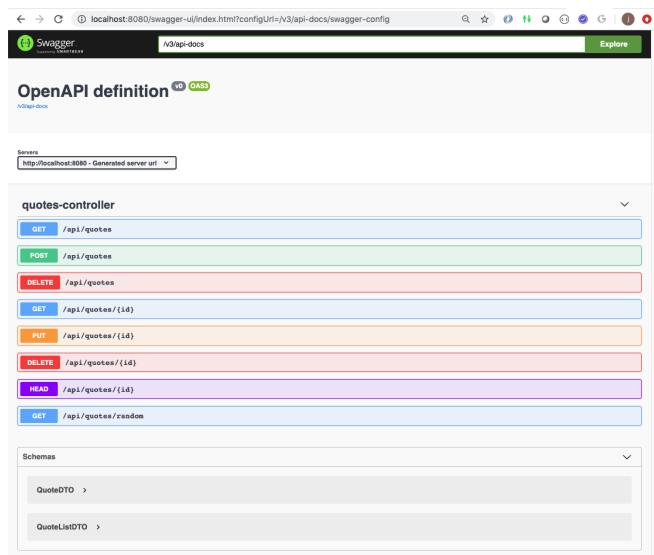
We will focus on using the Spring Boot 3 version, but the basics are pretty much the same with Spring Boot 2.

173.1.2. Springdoc Access

Once that is in place, you can access

- Open API spec: <http://localhost:8080/v3/api-docs>
- Swagger UI: <http://localhost:8080/swagger-ui.html>

The minimally configured Springdoc automatically provides an interface for our exposed web API.



173.1.3. Springdoc Starting Customization

The starting point for adjusting the overall interface for Springdoc is done through the definition of one or more GroupedOpenApi objects. From here, we can control the path and [countless other options](#). The specific option shown will reduce the operations shown to those that have paths that start with "/api/".

Springdoc Starting Customization

```
...
import org.springdoc.core.models.GroupedOpenApi;
import org.springdoc.core.utils.SpringDocUtils;

@Configuration
public class SwaggerConfig {
    @Bean
    public GroupedOpenApi api() {
        SpringDocUtils.getConfig();
        // ...
        return GroupedOpenApi.builder()
            .group("contests")
            .pathsToMatch("/api/**")
            .build();
    }
}
```

Textual descriptions are primarily added to the annotations of each controller and model/DTO class.

Chapter 174. Example Use

By this point in time we are past-ready for a live demo. You are invited to start the Springdoc version of the Contests Application and poke around. The following commands are being run from the parent `swagger-contest-example` directory. They can also be run within the IDE.

Start Springdoc Demo App

```
$ mvn spring-boot:run -f springdoc-contest-svc \①
-Dspring-boot.run.arguments==server.port=8080 ②
```

① option to use custom port number

② passes arguments from command line, though Maven, to the Spring Boot application

Access the application using

- Springdoc: <http://localhost:8080/swagger-ui.html>

I will show an example thread.

174.1. Access Contest Controller POST Command

- click on the `POST /api/contests` line and the details of the operation will be displayed.
- select a content type (`application/json` or `application/xml`)
- click on the "Try it out" button.

The screenshot shows the Springdoc UI for the `contest-controller`. The `POST /api/contests` endpoint is highlighted. The 'Parameters' section shows a parameter `newContest` with a description: 'The new contest to be added.' Below it is an 'Example Value' code block containing JSON:

```
{
  "id": 0,
  "homeTeam": "string",
  "awayTeam": "string",
  "scheduledStart": "2020-06-16T21:01:39.925Z",
  "duration": "PT0M0S",
  "completed": true,
  "homeScore": 2,
  "awayScore": 1
}
```

The 'Responses' section shows a 'curl' command:

```
curl -X POST "http://localhost:8080/api/contests" -H "Accept: application/xml" -H "Content-Type: application/xml" -d "<?xml version='1.0'?><ContestDTO><id></id><homeTeam>string</homeTeam><awayTeam>string</awayTeam><scheduledStart>2020-06-16T21:01:39.925Z</scheduledStart><duration>PT0M0S</duration><completed>true</completed><homeScore>2</homeScore><awayScore>1</awayScore></ContestDTO>"
```

The 'Server response' section shows a 201 status code with a response body containing XML:

```
<Contest xmlns="urn:java:svc-swagger.contests" id="21">
<scheduledStart>2020-06-16T21:01:39.936Z</scheduledStart>
<awayTeam>string</awayTeam>
<completed>true</completed>
<homeTeam>string</homeTeam>
<duration>PT0M0S</duration>
<homeScore>2</homeScore>
<awayScore>1</awayScore>
</Contest>
```

The 'Response headers' section shows:

```
connection: keep-alive
content-type: application/xml
date: Tue, 16 Jun 2020 21:03:25 GMT
last-modified: Tue, 16 Jun 2020 21:03:25 GMT
location: http://localhost:8080/api/contests/21
transfer-encoding: chunked
```

174.2. Invoke Contest Controller POST Command

- Overwrite the values of the example with your values.
- Select the desired response content type ([application/json](#) or [application/xml](#))
- press "Execute" to invoke the command

POST /api/contests This endpoint will create a new contest. Home and Away teams are required, id is ignored, and most other fields are optional.

Parameters

Name	Description
newContest (body)	The new contest to be added.

Example Value | Model

```
{
  "id": 0,
  "homeTeam": "string",
  "awayTeam": "string",
  "scheduledStart": "2020-06-16T21:01:39.925Z",
  "duration": "PT60M",
  "completed": true,
  "homeScore": 2,
  "awayScore": 1
}
```

Cancel

Parameter content type
application/json

Execute (Circled)
Clear

174.3. View Contest Controller POST Command Results

- look below the "Execute" button to view the results of the command. There will be a payload and some response headers.
- notice the payload returned will have an ID assigned
- notice the headers returned will have a location header with a URL to the created contest
- if your payload was missing a required field (e.g., home or away team), a 422/UNPROCESSABLE_ENTITY status is returned with a message payload containing the error text.

Responses

Curl

```
curl -X POST "http://localhost:8081/api/contests" -H "Accept: application/xml" -H "Content-Type: application/json" -d "({\"id\": 0, \"homeTeam\": \"string\", \"awayTeam\": \"string\", \"scheduledStart\": \"2020-06-16T21:01:39.925Z\", \"duration\": \"PT60M\", \"completed\": true, \"homeScore\": 2, \"awayScore\": 1})"
```

Request URL

<http://localhost:8081/api/contests>

Server response

Code	Details
201	Response body <pre><contest xmlns="urn:java:svc-swagger.contests" id="24"> <scheduledStart valns="">2020-06-16T21:01:39.925Z</scheduledStart> <duration valns="">PT60M</duration> <completed valns="">true</completed> <homeTeam valns="">string</homeTeam> <awayTeam valns="">string</awayTeam> <homeScore valns="">2</homeScore> <awayScore valns="">1</awayScore> </contest></pre> <p>Download</p>

Response headers

```
connection: keep-alive
content-type: application/xml
date: Tue, 16 Jun 2020 21:58:41 GMT
keep-alive: timeout=60
location: http://localhost:8081/api/contests/24
transfer-encoding: chunked
```

Chapter 175. Useful Configurations

I have created a set of examples under the [Swagger Contest Example](#) that provide a significant amount of annotations to add descriptions, provide accurate responses to dynamic outcomes, etc. using Springdoc to get a sense of how they performed.

The screenshot shows the Swagger UI interface for a Springdoc application. At the top, there's a navigation bar with the Swagger logo, a dropdown menu labeled "Select a definition" containing "contests", and a version indicator "v1 OAS3". Below the header, the title "Springdoc Swagger Contest Example" is displayed, along with a link to "v1 OAS3". A brief description states: "This application provides an example of how to provide extra swagger and springfox configuration in order to better document the API." There are links to "Jim Stafford - Website" and "http://localhost:8082 - Generated server url".

The main content area is titled "contest-controller". It lists various HTTP methods and their corresponding endpoints:

- GET /api/contests**: This endpoint will return a collection of contests based on the paging values assigned.
- POST /api/contests**: This endpoint will create a new contest. Home and Away teams are required, id is ignored, and most other fields are optional.
- DELETE /api/contests**: This method will delete all contests.
- GET /api/contests/{contestId}**: Returns the specified contest.
- PUT /api/contests/{contestId}**: Update an existing contest -- perhaps to schedule it or report scores.
- DELETE /api/contests/{contestId}**: This method will delete the specified contest. There is no error if the ID does not exist. There will only be an error when we fail to delete it.
- HEAD /api/contests/{contestId}**: This method simply checks whether the contest exists.

Below the controller section, there's a "Schemas" section showing the definitions for "ContestDTO" and "ContestListDTO".

ContestDTO (grey background):

```
ContestDTO >
```

MessageDTO (grey background):

```
MessageDTO ▾ {  
    description:  
        This class is used to convey a generic message -- usually the result of an error.  
    url  
        string  
        The URL that generated the message.  
    text  
        string  
        The text of the message.  
    date  
        string($date-time)  
        The date (UTC) of the generated message.  
}
```

ContestListDTO (grey background):

```
ContestListDTO >
```

Figure 59. Fully Configured Springdoc Example

That is a lot of detail work and too much to cover here for what we are looking for. Feel free to look at the examples ([controller](#), [dtos](#)) for details. However, I did encounter a required modification that made a feature go from unusable to usable and will show you that customization in order to give you a sense of how you might add other changes.

175.1. Customizing Type Expressions

`java.time.Duration` has a simple [ISO string format](#) expression that looks like `PT60M` or `PT3H` for periods of time.

175.1.1. OpenAPI 3 Model Property Annotations

The following snippet shows the Duration property enhanced with Open API 3 annotations, without any rendering hints.

ContestDTO Snippet with Duration without Rendering Hints

```
package info.ejava.examples.svc.springdoc.contests.dto;

import com.fasterxml.jackson.annotation.JsonProperty;
import com.fasterxml.jackson.dataformat.xml.annotation.JacksonXmlProperty;
import com.fasterxml.jackson.dataformat.xml.annotation.JacksonXmlRootElement;
import io.swagger.v3.oas.annotations.media.Schema;

@JacksonXmlRootElement(localName="contest", namespace=ContestDTO.CONTEST_NAMESPACE)
@Schema(description="This class describes a contest between a home and away team, "+
    "either in the past or future.")
public class ContestDTO {
    @JsonProperty(required = false)
    @Schema(
        description="Each scheduled contest should have a period of time specified "+
            "that identifies the duration of the contest. e.g., PT60M, PT2H")
    private Duration duration;
```

175.1.2. Default Duration Example Renderings

Springdoc's example rendering for `java.util.Duration` for JSON and XML is not desirable. It is basically trying to render an unknown type using reflection of the `Duration` class.

Springdoc Default Duration JSON Expression

```
"duration": {  
    "seconds": 0,  
    "nano": 0,  
    "negative": true,  
    "zero": true,  
    "units": [  
        {  
            "dateBased": true,  
            "timeBased": true,  
            "duration": {  
                "seconds": 0,  
                "nano": 0,  
                "negative": true,  
                "zero": true  
            },  
            "durationEstimated": true  
        }  
    ]  
}
```

Springdoc Default Duration XML Expression

```
<duration>  
    <seconds>0</seconds>  
    <negative>true</negative>  
    <zero>true</zero>  
    <units>  
        <durationEstimated>  
        true</durationEstimated>  
        <duration>  
            <seconds>0</seconds>  
            <negative>true</negative>  
            <zero>true</zero>  
            <nano>0</nano>  
        </duration>  
        <dateBased>true</dateBased>  
        <timeBased>true</timeBased>  
    </units>  
    <nano>0</nano>  
</duration>
```

175.1.3. OpenAPI 3 Model Property Annotations

We can add an `example` to the `@Schema` annotation to override the produced value. Here we are expressing a `PT60M` example value be used.

ContestDTO Snippet with Duration and OpenAPI 3 Annotations

```
@JsonProperty(required = false)  
@Schema(  
    example = "PT60M",  
    description="Each scheduled contest should have a period of time specified "+  
               "that identifies the duration of the contest. e.g., PT60M, PT2H")  
private Duration duration;
```

175.1.4. Simple Duration Example Renderings

The `example` attribute worked for JSON but did not render well for XML. The example text was displayed for XML without its `<duration></duration>` tags.

Springdoc-wide Duration Mapped to String JSON Expression

```
{  
    "id": 0,  
    "scheduledStart": "2023-08-  
02T12:48:29.769Z",  
    "duration": "PT60M", ①  
    "completed": true,  
    "homeTeam": "string",  
    "awayTeam": "string",  
    "homeScore": 2,  
    "awayScore": 1  
}
```

① correctly rendered provided PT60M example

Springdoc Property-specific Duration Mapped to String XML Expression

```
<?xml version="1.0" encoding="UTF-8"?>  
<ContestDTO>  
    <scheduledStart>2023-08-  
02T13:36:20.867Z</scheduledStart>  
    <completed>true</completed> ①  
    <homeTeam>string</homeTeam>  
    <awayTeam>string</awayTeam>  
    <homeScore>2</homeScore>  
    <awayScore>1</awayScore>  
</ContestDTO>
```

① incorrectly rendered XML for duration element

175.1.5. Schema Sub-Type Renderings

I was able to correct JSON and XML examples using a "StringSchema example". The following example snippets show expressing the schema at the property and global level.

Example Springdoc Map Property-specific to Schema Example

```
@JsonProperty(required = false)  
@Schema("//position = 4,  
        example = "PT60M",  
        type = "string", ①  
        description = "Each scheduled contest should have a period of time specified  
that " +  
                "identifies the duration of the contest. e.g., PT60M, PT2H")  
private Duration duration;
```

① type allows us to subtype **Schema** for string expression

A global override can be configured at application startup.

Example Springdoc-wide Map Class to Schema Example

```
SpringDocUtils.getConfig()  
    .replaceWithSchema(Duration.class,  
        new StringSchema().example("PT120M")); ①
```

① **StringSchema** is a subclass of **Schema**

The **subtypes** of **Schema** were important to get both the JSON and XML rendered appropriately.

175.1.6. StringSchema Duration Example Renderings

The examples below show the payloads expressed in a usable form and potentially a valuable

source default example. Global configuration overrides property-specific configuration if both are used.

Springdoc-wide Duration Mapped to String JSON Expression

```
{  
    "id": 0,  
    "scheduledStart": "2023-08-  
02T12:48:29.769Z",  
    "duration": "PT120M",  
    "completed": true,  
    "homeTeam": "string",  
    "awayTeam": "string",  
    "homeScore": 2,  
    "awayScore": 1  
}
```

Springdoc Property-specific Duration Mapped to String XML Expression

```
<?xml version="1.0" encoding="UTF-8"?>  
<ContestDTO>  
    <scheduledStart>2023-08-  
02T13:36:20.867Z</scheduledStart>  
    <duration>PT60M</duration>  
    <completed>true</completed>  
    <homeTeam>string</homeTeam>  
    <awayTeam>string</awayTeam>  
    <homeScore>2</homeScore>  
    <awayScore>1</awayScore>  
</ContestDTO>
```

The example explored above was good enough to get my quick example usable, but shows how common it can be to encounter a bad example rendered document. Swagger-core has many options to address this that could be an entire set of lectures itself. The basic message is that Swagger provides the means to express usable examples to users, but you have to dig.

Chapter 176. Client Generation

We have seen where the OpenAPI schema definition was used to generate a UI rendering. We can also use it to generate a client. I won't be covering any details here, but will state there is a [generation example](#) in the class examples that produces a Java jar that can be used to communicate with our server—based in the OpenAPI definition produced. The example uses the [OpenAPI Maven Plugin](#) that looks extremely similar to the [Swagger CodeGen Maven Plugin](#). There repository page has a ton of configuration options. This example uses just the basics.

176.1. API Schema Definition

The first step should be to capture the schema definition from the server into a CM artifact. Running locally, this would come from <http://localhost:8080/v3/api-docs/contests>.

OpenAPI Interface Definition File

```
{
  "openapi": "3.0.1",
  "info": {
    "title": "Springdoc Swagger Contest Example",
    "description": "This application provides an example of how to provide extra swagger and springfox configuration in order to better document the API.",
    "contact": {
    },
    "version": "v1"
  },
  "servers": [
    {
      "url": "http://localhost:8080",
      "description": "Generated server url"
    }
  ],
  "tags": [
    {
      "name": "contest-controller",
      "description": "manages contests"
    }
  ],
  "paths": {
    "/api/contests/{contestId}": {
      ...
    }
  }
}
```

176.2. API Client Source Tree

We can capture that file in a module source tree used for generation.

```
|-- pom.xml  
|-- src  
  '-- main  
    '-- resources  
      '-- contests-api.json
```

176.3. OpenAPI Maven Plugin

We add the Maven plugin to compile the API schema definition. The following shows a minimal skeletal plugin definition (version supplied by parent). Refer to the plugin web pages for options that can control building the source.

OpenAPI Maven Plugin

```
<plugin>  
  <groupId>org.openapitools</groupId>  
  <artifactId>openapi-generator-maven-plugin</artifactId>  
  <executions>  
    <execution>  
      <goals>  
        <goal>generate</goal>  
      </goals>  
      <configuration>  
        <inputSpec>${project.basedir}/src/main/resources/contests-  
api.json</inputSpec>  
        <generatorName>java</generatorName>  
        <configOptions>  
          <sourceFolder>src/gen/java/main</sourceFolder>  
        </configOptions>  
      </configuration>  
    </execution>  
  </executions>  
</plugin>
```

176.4. OpenAPI Generated Target Tree

With the plugin declared, we can execute `mvn clean generate-sources`, to get a generated source tree with Java files containing some familiar names—like `model.ContestDTO`, `ContestListDTO`, and `MessageDTO`.

OpenAPI Generated Target Tree

```
'-- target  
  '-- generated-sources  
    '-- openapi  
      |-- README.md
```

```

...
`-- src
  |-- gen
  |  '-- java
  |    '-- main
  |      '-- org
  |        '-- openapitools
  |          '-- client
  |            |-- ApiCallback.java
...
...
  '-- model
    |-- ContestDTO.java
    |-- ContestListDTO.java
    '-- MessageDTO.java

```

176.5. OpenAPI Compilation Dependencies

To compile the generated source, we are going to have to add some dependencies to the module. In my quick read through this capability, I was surprised that the dependencies were not more obviously identified and easier to add. The following snippet shows my result of manually resolving all compilation dependencies.

```

<!-- com.google.gson -->
<dependency>
  <groupId>io.gsonfire</groupId>
  <artifactId>gson-fire</artifactId>
</dependency>

<!-- javax.annotation.Nullable -->
<dependency>
  <groupId>com.google.code.findbugs</groupId>
  <artifactId>jsr305</artifactId>
</dependency>

<!-- javax.annotation.Generated -->
<dependency>
  <groupId>javax.annotation</groupId>
  <artifactId>javax.annotation-api</artifactId>
</dependency>

<!-- okhttp3.internal.http.HttpMethod -->
<dependency>
  <groupId>com.squareup.okhttp3</groupId>
  <artifactId>okhttp</artifactId>
</dependency>

<!-- okhttp3.logging.HttpLoggingInterceptor -->
<dependency>
  <groupId>com.squareup.okhttp3</groupId>

```

```
<artifactId>logging-interceptor</artifactId>
</dependency>
```

176.6. OpenAPI Client Build

With API definition in place, plugin defined and declared, and dependencies added, we can now generate the client JAR.

```
$ mvn clean install
...
[INFO] --- openapi-generator-maven-plugin:7.8.0:generate (default) @ generated-
contest-client ---
[INFO] Generating with dryRun=false
[INFO] OpenAPI Generator: java (client)
...
[INFO] Processing operation getContest
[INFO] Processing operation updateContest
[INFO] Processing operation doesContestExist
[INFO] Processing operation deleteContest
[INFO] Processing operation getContests
[INFO] Processing operation createContest
[INFO] Processing operation deleteAllContests
[INFO] writing file .../svc/svc-api/swagger-contest-example/generated-contest-
client/target/generated-
sources/openapi/src/gen/java/main/org/openapitools/client/model/ContestDTO.java
...
#####
# Thanks for using OpenAPI Generator. #
# Please consider donation to help us maintain this project ☺ #
# https://opencollective.com/openapi_generator/donate #
#####
...
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 6.814 s
```

This was light coverage of the OpenAPI client generation capability. It gives you insight into why OpenAPI is useful for building external clients as well as the Swagger UI capability.

Chapter 177. Springdoc Summary

Springdoc is surprisingly functional right out of the box with minimal configuration—except for some complex types. In early June 2020, Springdoc definitely understood the purpose of the Java code better than Springfox and quickly overtook Springfox before the older option eventually stopped turning out releases. Migrating Springdoc from Spring Boot 2 to 3 was primarily a Maven dependency and a few package name changes for their configuration classes.

It took some digging, but I was able to find a solution to my specific `Duration` rendering problem. Along the way, I saw examples of how I could provide example payloads and complex objects (with values) that could be rendered into example payloads. A custom example is quite helpful if the model class has a lot of optional fields that are rarely used and unlikely to be used by someone using the Swagger UI.

[Springfox has better documentation](#) that shows you features ahead of time in logical order. [Springdoc's documentation](#) is primarily a Q&A FAQ that shows features in random order. I could not locate a good Springdoc example when I got started—but after implementing with Springfox first, the translation was extremely easy. Following the provided example in this lecture should provide you with a good starting point.

Springfox had been around a long time but with the change from Open API 2 to 3, the addition of Webflux, and their slow rate of making changes—that library soon showed to not be a good choice for Open API or Webflux users. Springdoc seemed like it was having some early learning pains in 2020—where features may have worked easier but didn't always work 100%, lack of documentation and examples to help correct, and their existing FAQ samples did not always match the code. However, it was quite usable already in early versions, already supports Spring Boot 3 in 2023, and they continue to issuing releases. By the time you read this much may have changed.

One thing I found after adding annotations for the technical frameworks (e.g., Lombok, WebMVC, Jackson JSON, Jackson XML) and then trying to document every corner of the API for Swagger in order to flesh out issues—it was hard to locate the actual code. My recommendation is to continue to make the good names for controllers, models/DTO classes, parameters, and properties that are immediately understandable to save on the extra overhead of Open API annotations. Skip the obvious descriptions one can derive from the name and type, but still make it document the interface and usable to developers learning your API.

Chapter 178. Summary

In this module we:

- learned that Swagger is a landscape of items geared at delivering HTTP-based APIs
- learned that the company Smartbear originated Swagger and then divided up the landscape into a standard interface, open source tools, and commercial tools
- learned that the Swagger standard interface was released to open source at version 2 and is now Open API version 3
- learned that two tools—Springfox and Springdoc—were focused on implementing Open API for Spring and Spring Boot applications and provided a packaging of the Swagger UI
- learned that Springfox and Springdoc have no formal ties to Spring, Spring Boot, Pivotal, Smartbear, etc. They are their own toolset and are not as polished (in 2020) as we have come to expect from the Spring suite of libraries.
- learned that Springfox is older, originally supported Open API 2 and SpringMVC for many years, and now supports Open API 3, Spring Boot 2, and WebFlux. Springfox has stopped producing releases since July 2020.
- learned that Springdoc is newer, active, and supports Open API 3, SpringMVC, Webflux, and Spring Boot 3
- learned how to minimally configure Springdoc into our web application in order to provide the simple ability to invoke our HTTP endpoint operations
- learned how to minimally setup API generation using the OpenAPI schema definition from a running application and a Maven plugin

<groupId>info.ejava.assignments.api.autorentals</groupId>

AutoRentals Assignment 2: API

copyright © 2024 jim stafford (jim.stafford@jhu.edu)

The API assignment is a single assignment that is broken into focus areas that relate 1:1 with the lecture topics. You have the choice to start with any one area and either advance or jump repeatedly between them as you complete the overall solution. However, you are likely going to want to start out with the modules area so that you have some concrete modules to begin your early work. It is always good to be able to perform a successful root level build of all targeted modules before you begin adding detailed dependencies, plugins, and Java code.

Chapter 179. Overview

The API will include three main concepts. We are going to try to keep the business rules pretty simple at this point:

1. **Auto** - an individual auto that can be part of an AutoRental
 - a. Autos can be created, modified, listed, and deleted
2. **Renter** - identification for a person that may be part of a AutoRental and is not specific to any one AutoRental
 - a. Renters can be created, modified, listed, and deleted
3. **AutoRental** - identifies a specific Auto and Renter agreement for a period of time
 - a. AutoRentals must be created with an existing auto and renter
 - b. AutoRentals must be created with a current or future, non-overlapping period of time for the auto
 - c. AutoRentals must be created with a renter with a minimum age (21)
 - d. AutoRental time span is modifiable as long as the auto is available
 - e. AutoRental can be deleted

179.1. Grading Emphasis

Grading emphasis is focused on the demonstration of satisfying the listed learning objectives and—except the scenarios listed at the end—not on quantity. Most required capability/testing is focused on demonstration of what you know how to do. You are free to implement as much of the business model as you wish, but treat completing the listed scenarios at the end of the assignment (within the guidance of the stated static requirements) as the minimal functionality required.

179.2. AutoRenter Support

You are given a complete implementation of Auto and Renter as examples and building blocks in order to complete the assignment. Your primary work will be in completing AutoRentals.

179.2.1. AutoRenter Service

The `autorenters-support-api-svc` module contains a full @RestController/Service/Repo thread for both Autos and Renters. The module contains two Auto-configuration definitions that will automatically activate and configure the two services within a dependent application.

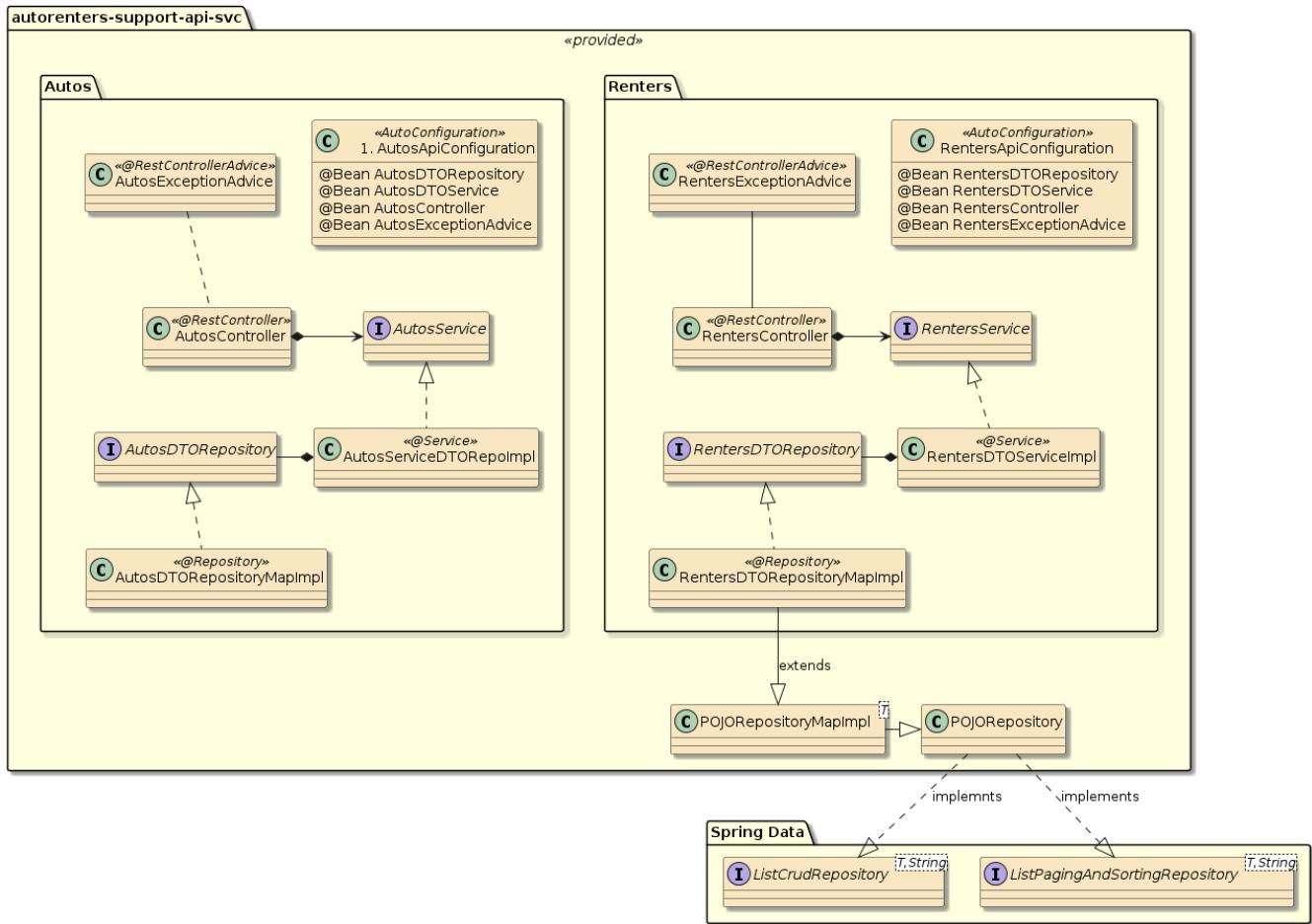


Figure 60. autorenters-support-api-svc Module

The following dependency can be added to your service solution to bring in the Autos and Renters service examples to build upon.

AutoRenter Service Dependency

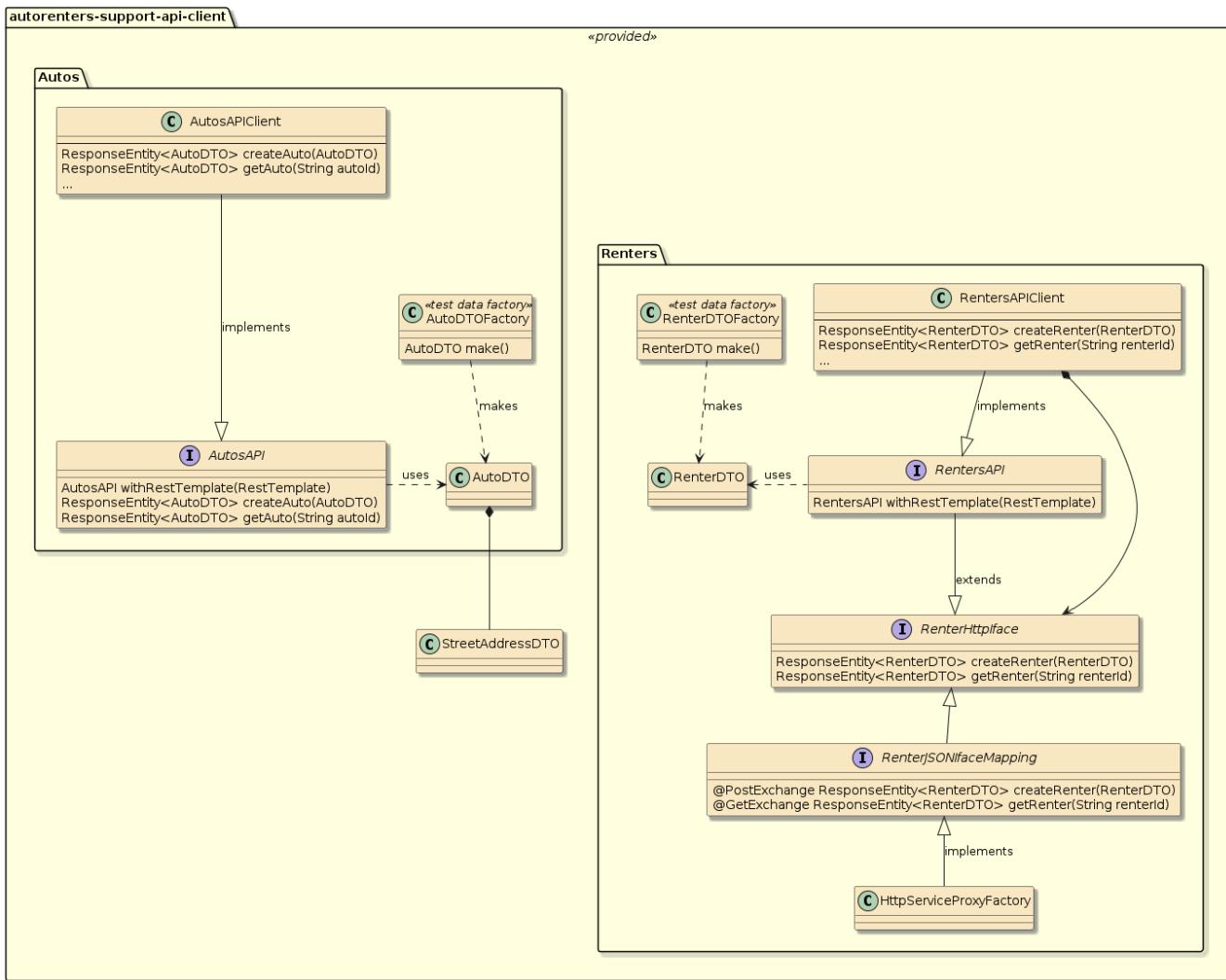
```
<dependency>
    <groupId>info.ejava.assignments.api.autorentals</groupId>
    <artifactId>autorenters-support-api-svc</artifactId>
    <version>${ejava.version}</version>
</dependency>
```

179.2.2. AutoRenter Client

A client module is supplied that includes the DTOs and client to conveniently communicate with the APIs. Your AutoRental solution will inject the Autos and Renters service components for interaction but your API tests will use the Auto and Renter http-based APIs. For demonstration,

- the AutoAPIClient is implemented using explicit RestTemplate calls
- the RenterAPIClient is implemented using Spring 6 Http Interface

The implementation details are available for inspection, but encapsulated such that they work the same via their AutosAPI and RentersAPI.



The following dependency can be added to your solution to bring in the Autos and Renters client artifact examples to build upon.

AutoRenter Client Dependency

```

<dependency>
    <artifactId>autorenters-support-api-client</artifactId> ①
    <version>${ejava.version}</version>
</dependency>

```

① dependency on client will bring in both client and dto modules

179.2.3. AutoRenter Tests

You are also supplied a set of tests that are meant to assist in your early development of the end-to-end capability. You are still encouraged to write your own tests and required to do so in specific sections and for the required scenarios.



You may find it productive and helpful to address the supplied tests first and implement the scenario tests at the end. It is up to you. At any time you can break off and write special-purpose tests for your own purpose.

The supplied tests are made available to you using the following Maven dependency.

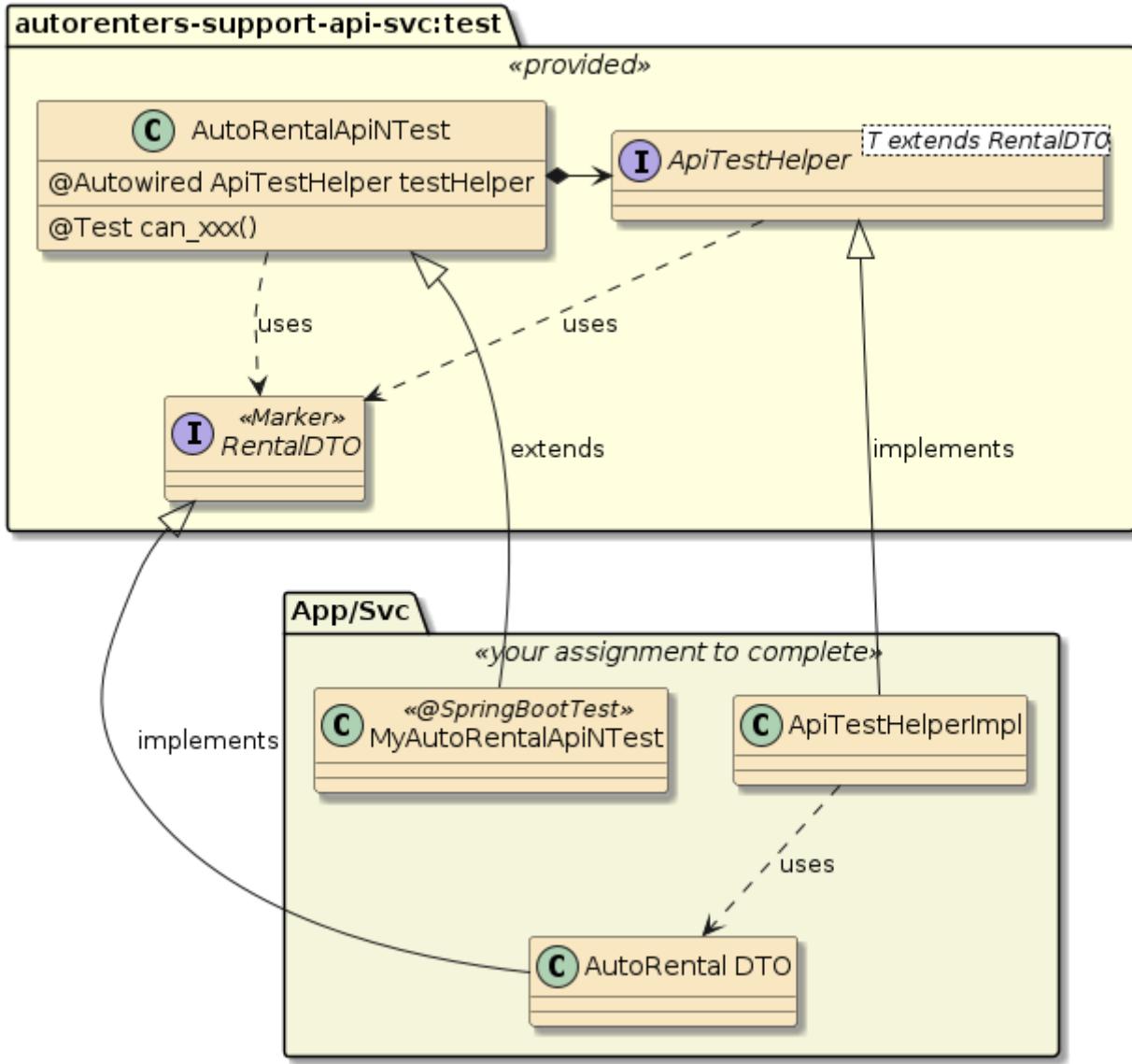
AutoRenter Tests dependency

```
<dependency>
    <groupId>info.ejava.assignments.api.autorentals</groupId>
    <artifactId>autorenters-support-api-svc</artifactId>
    <version>${ejava.version}</version>
    <classifier>tests</classifier> ①
    <scope>test</scope>
</dependency>
```

① tests have been packaged within a separate `-tests.java`

The provided tests require that:

- your AutoRental DTO class implement a `RentalDTO` "marker" interface provided by the support module. This interface has nothing defined and is only used to identify your DTO during the tests.
- implement a `ApiTestHelper<T extends RentalDTO>` interface and make it a component available to be injected into the provided tests. A full skeleton of this class implementation has been supplied in the starter.
- supply a `@SpringBootTest` class that pulls in the `AutoRentalsApiITest` test case as a base class from the support module. This test case evaluates your solution during several core steps of the assignment. Much of the skeletal boilerplate for this work is provided in the starter.



Enable the tests whenever you are ready to use them. This can be immediately or at the end.

Groups of Tests can be Disabled Early In Development

There are empty `@Nested` classes provided in your starter's `MyAutoRentalApiNTTest` that hide a group of tests in the parent while you focus on another group. Comment the `@Nested` declaration to enable a new group of tests.



```
//remove this class declaration to enable query tests
@Nested public class can_query_to { }
```

The tests are written to execute from the subclass in your area. With adhoc navigation, sometimes the IDE can get lost—lose the context of the subclass and provide errors as if there were only the base class. If that occurs—make a more direct IDE command to run the subclass to clear the issue.



Chapter 180. Assignment 2a: Modules

180.1. Purpose

In this portion of the assignment, you will demonstrate your knowledge of establishing Maven modules for different portions of an application. You will:

1. package your implementation along proper module boundaries

180.2. Overview

In this portion of the assignment you will be establishing your source module(s) for development. Your new work should be spread between two modules:

- a single client module for DTO and other API artifacts
- a single application module where the Spring Boot executable JAR is built

Your client module should declare a dependency on the provided `autorenters-support-api-client` to be able to make use of any DTO or API constructs. Your service/App module should declare a dependency on `autorenters-support-api-svc` to be able to host the Auto and Renter services. You do not copy or clone these "support" modules. Create a Maven dependency on these and use them as delivered.

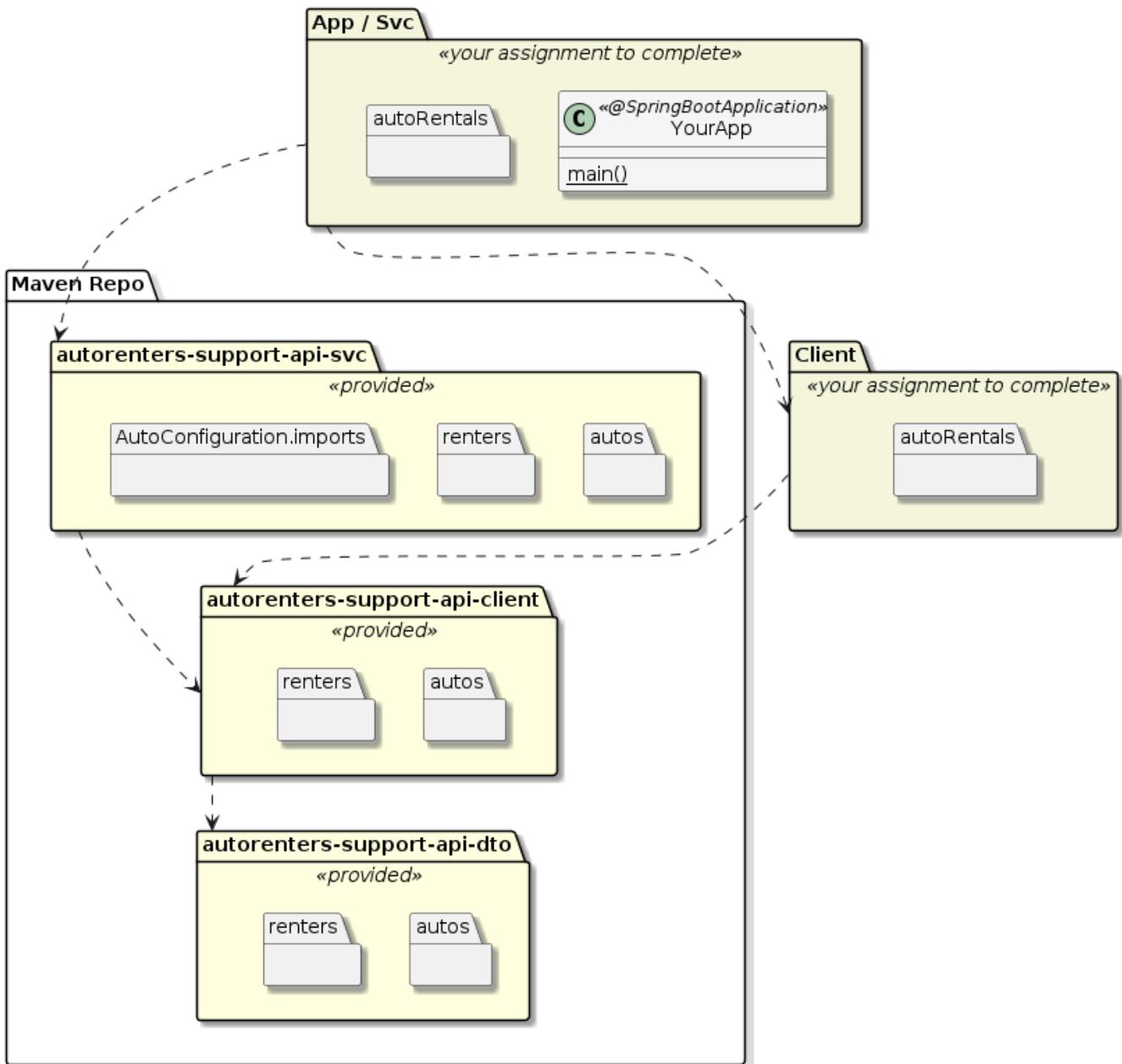


Figure 61. Module Packaging

180.3. Requirements

1. Create your overall project as two (or more) Maven modules under a single parent
 - a. **client** module(s) should contain any dependencies required by a client of the Web API. This includes the DTOs, any helpers created to implement the API calls, and unit tests for the DTOs. This module produces a regular Java JAR. **autorenters-support-api-client/dto** has been supplied for you use as an example and be part of your client modules. Create a dependency on the client module for access to **Auto** and **Renter** client classes. Do not copy/clone the support modules.
 - b. **svc** module to include your AutoRentals controller, service, and repository work. **autorenters-support-api-svc** has been supplied for you to both be part of your solution and to use as an example. Create a Maven dependency on this support module. Do not copy/clone it.
 - c. **app** module that contains the **@SpringBootApplication** class will produce a Spring Boot

Executable JAR to instantiate the implemented services.



The **app** and **svc** modules can be the **same module**. In this dual role, it will contain your AutoRental service solution and also host the `@SpringBootApplication`.



The Maven pom.xml in the assignment starter for the App builds both a standard library JAR and a separate executable JAR (bootexec) to make sure we retain the ability to offer the AutoRental service as a library to a downstream assignment. By following this approach, you can make this assignment immediately reusable in assignment 3.

- d. **parent** module that establishes a common groupId and version for the child modules and delegate build commands. This can be the same parent used for assignments 0 and 1. Only your app and client modules will be children of this parent.
2. Define the svc module as a Web Application (dependency on `spring-boot-starter-web`).
3. Add a `@SpringBootApplication` class to the app module (*already provided in starter for initial demo*).
4. Once constructed, the modules should be able to
 - a. build the project from the root level
 - b. build regular Java JARs for use in downstream modules
 - c. build a Spring Boot Executable JAR (bootexec) for the `@SpringBootApplication` module
 - d. immediately be able to access the `/api/autos` and `/api/renters` resource API when the application is running — because of Auto-Configuration.

Example Calls to Autos and Renters Resource API

```
$ curl -X GET http://localhost:8080/api/autos
{"contents":[]}
$ curl -X GET http://localhost:8080/api/renters
{"contents":[]}
```

180.4. Grading

Your solution will be evaluated on:

1. package your implementation along proper module boundaries
 - a. whether you have divided your solution into separate module boundaries
 - b. whether you have created appropriate dependencies between modules
 - c. whether your project builds from the root level module
 - d. whether you have successfully activated the Auto and Renter API

180.5. Additional Details

1. Pick a Maven hierarchical groupId for your modules that is unique to your overall work on this assignment.

Chapter 181. Assignment 2b: Content

181.1. Purpose

In this portion of the assignment, you will demonstrate your knowledge of designing a Data Transfer Object that is to be marshalled/unmarshalled using various internet content standards. You will:

1. design a set of Data Transfer Objects (DTOs) to render information from and to the service
2. define a Java class content type mappings to customize marshalling/unmarshalling
3. specify content types consumed and produced by a controller
4. specify content types accepted by a client

181.2. Overview

In this portion of the assignment you will be implementing a set of DTO classes that will be used to represent a AutoRental. All information expressed in the AutoRental will be derived from the Auto and Renter objects — except for the ID and the milestone properties.



Lecture/Assignment Module Ordering

It is helpful to have a data model in place before writing your services. However, the lectures are structured with a content-less (String) domain up front — focusing on the Web API and services before tackling content. If you are starting this portion of the assignment before we have covered the details of content, it is suggested that you simply create sparsely populated AutoRentalDTO class with at least an `id` field and the AutoRentalListDTO class to be able to complete the API interfaces. Skip the details of this section until we have covered the Web content lecture.

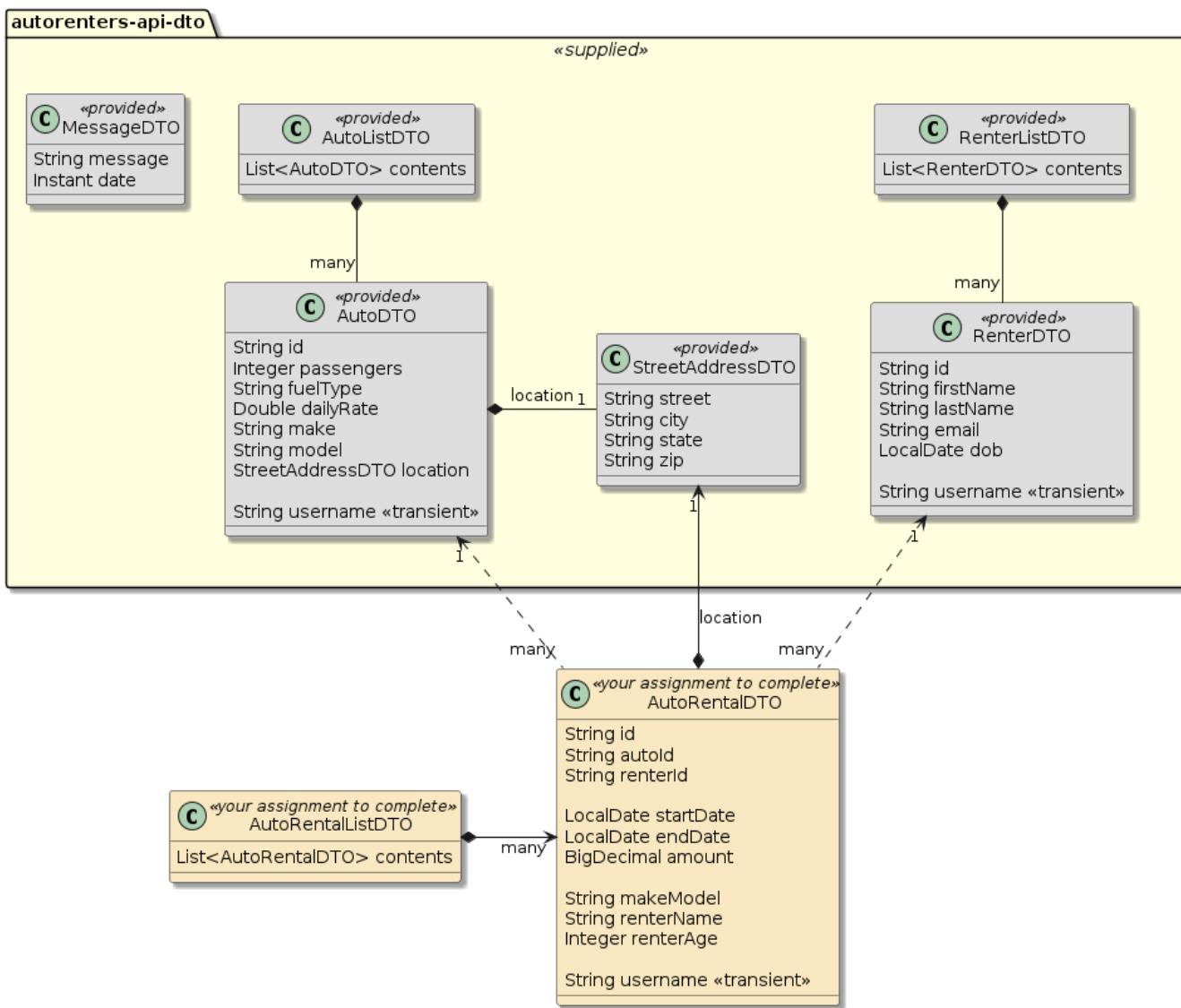


Figure 62. Content

AutoRental.id Avoids Compound Primary Key



The AutoRentalDTO **id** was added to keep from having to use a compound (autoId + renterId) primary key. This makes it an easier 1:1 example with Auto and Renter to follow.

String Primary Keys



Strings were used for the primary key type. This will make it much easier and more portable when we use database repositories in a later assignment.

The provided `autorenters-support-api-dto` module has the Auto and Renter DTO classes.

- AutoDTO - provides information specific to the auto
- RenterDTO - provides information specific to the renter
- StreetAddress - provides properties specific to a location for the Auto
- MessageDTO - commonly used to provide error message information for request failures
- <Type>ListDTO - used used to conveniently express typed collection of objects

MessageDTO is from ejava-dto-util Class Examples



A **MessageDTO** is supplied in the **ejava-dto-util** package and used in most of the class API examples. You are free to create your own for use with the AutoRentals portion of the assignment.

181.3. Requirements

1. Create a DTO class to represent AutoRental. Use the attributes in the diagram above and descriptions below as candidate properties for each class.
 - a. The following attributes can be expressed both client and server-side
 - i. `autoId` - required and an external reference to `Auto`
 - ii. `renterId` - required and an external reference to `Renter`
 - iii. `startDate` - required can be before or equal to `endDate`
 - iv. `endDate` - required and must be after or equal to `startDate`

TimeSpan Class



I have provided a **TimeSpan** utility class that can be used as an optional compound encapsulation of `startDate` and `endDate` when accessing properties of the AutoRental DTO. It can also be optionally leveraged for time span overlap comparisons.

- b. The following attributes are only assigned server-side.
 - i. `id` is a unique ID for a AutoRental
 - ii. `makeModel` should be a concatenation of `Auto.make` and `Auto.model` such that an evaluation of the string will contain the original make and model string values
 - iii. `renterName` should be the concatenation of `Renter.firstName` and `Renter.lastName` such that an evaluation of the string will contain the original first and lastName string values
 - iv. `renterAge` should be a calculation of years, rounded down, between the `Renter.dob` and the date the AutoRental starts.
 - c. `streetAddress` should be a deep copy of the `Auto.location`
2. Create a `AutoRentalListDTO` class to provide a typed collection of `AutoRentalDTO`.



I am recommending you name the collection within the class a generic `contents` for later reuse reasons.

3. Map each DTO class to:
 - a. Jackson JSON (**the only required form**)
 - b. *mapping to Jackson XML is optional*
4. Create a unit test to verify your new DTO type(s) can be marshalled/unmarshalled to/from the targeted serialization type.
5. API TODO: Annotate controller methods to consume and produce supported content type(s)

when they are implemented.

6. API TODO: Update clients used in unit tests to explicitly only accept supported content type(s) when they are implemented.

181.4. Grading

Your solution will be evaluated on:

1. design a set of Data Transfer Objects (DTOs) to render information from and to the service
 - a. whether DTO class(es) represent the data requirements of the assignment
2. define a Java class content type mappings to customize marshalling/unmarshalling
 - a. whether unit test(s) successfully demonstrate the ability to marshall and unmarshal to/from a content format
3. API TODO: specify content types consumed and produced by a controller
 - a. whether controller methods are explicitly annotated with consumes and produces definitions for supported content type(s)
4. API TODO: specify content types accepted by a client
 - a. whether the clients in the unit integration tests have been configured to explicitly supply and accept supported content type(s).

181.5. Additional Details

1. This portion of the assignment alone primarily produces a set of information classes that make up the primary vocabulary of your API and service classes.
2. Use of `lombok` is highly encouraged here and can tremendously reduce the amount of code you write for these classes
3. The Java `Period` class can easily calculate age in years between two `LocalDates`. A `TimePeriod` class has been provided in the DTO package to aggregate startDate and endDate LocalDates together and to provide convenience methods related to those values.
4. The `autorenters-support-api-client` module also provides a `AutoDTOFactory`, `RenterDTOFactory`, and `StreetAddressDTOFactory` that makes it easy for tests and other demonstration code to quickly assemble example instances. You are encouraged to follow that pattern.
5. The `autorenters-support-api-client` test cases for Auto and Renter demonstrate marshalling and unmarshalling DTO classes within a JUnit test. You should create a similar test of your `AutoRenterDTO` class to satisfy the testing requirement. Note that those tests leverage a `JsonUtil` class that is part of the class utility examples and simplifies example use of the Jackson JSON parser.
6. The `autorenters-support-api-client` and supplied starter unit tests make use of JUnit `@ParameterizedTest`—which allows a single JUnit test method to be executed N times with variable parameters—*pretty cool feature*. Try it.
7. Supporting multiple content types is harder than it initially looks—especially when trying to mix different libraries. WebClient does not currently support Jackson XML and will attempt to

resort to using JAXB in the client. I provide an example of this later in the semester (Spring Data JPA End-to-End) and advise you to address the **optional** XML mapping last after all other requirements of the assignment are complete. If you do attempt to tackle **both** XML and WebClient together, know to use JacksonXML mappings for the server-side and JAXB mappings for the client-side.

Chapter 182. Assignment 2c: Resources

182.1. Purpose

In this portion of the assignment, you will demonstrate your knowledge of designing a simple Web API. You will:

1. identify resources
2. define a URI for a resource
3. define the proper method for a call against a resource
4. identify appropriate response code family and value to use in certain circumstances

182.2. Overview

In this portion of the assignment you will be identifying a resource to implement the AutoRental API. Your results will be documented in a @RestController class. There is nothing to test here until the DTO and service classes are implemented.

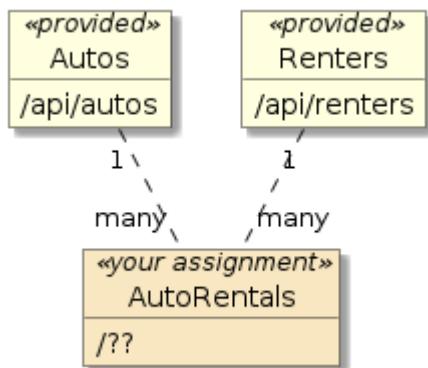


Figure 63. Identify Resources

The API will include three main concepts:

1. **Autos (provided)** - an individual auto that can be part of a auto rental
 - a. *Auto information can be created, modified, listed, and deleted*
 - b. *Auto information can be modified or deleted at any time but changes do not impact previous auto rentals*
2. **Renters (provided)** - identification for a person that may participate in a auto rental
 - a. *Renter information can be created, modified, listed, and deleted*
 - b. *Renter information can be modified or deleted at any time but changes do not impact previous auto rentals*
3. **AutoRentals** (your assignment) - a transaction for one Auto and one Renter for a unique period of time
 - a. *AutoRental information can be created, modified, listed, and deleted*

- b. business rules will be applied for new and modified AutoRentals

182.3. Requirements

Capture the expression of the following requirements in a set of `@RestController` class(es) to represent your resources, URIs, required methods, and status codes.

1. Identify your base resource(s) and sub-resource(s)

- a. create URIs to represent each resource and sub-resource

Example Skeletal API Definitions

```
public interface AutosAPI {  
    String AUTOS_PATH="/api/autos";  
    String AUTO_PATH="/api/autos/{id}";  
    ...
```

- b. create a separate `@RestController` class—at a minimum—for each base resource

Example Skeletal Controller

```
@RestController  
public class AutosController {
```

2. Identify the `@RestController` methods required to represent the following actions for AutoRental. Assign them specific URIs and HTTP methods.

- a. create new AutoRental resource (a "contract")

- i. Auto and Renter must exist
- ii. provided time period must be current or future dates
- iii. provided time period must not overlap with existing AutoRental for the same Auto for two different Renters (same Renter may have overlapping AutoRentals)

- iv. Renter must be at least 21 on startDate of rental

- b. update an existing AutoRental resource

- i. modified time period must be current or future dates
- ii. modified time period must not overlap with existing AutoRental for the same Auto for two different Renters

- c. get a specific AutoRental resource

- d. list AutoRental resources with paging

- i. accept optional autoId, renterId, and time period query parameters
- ii. accept optional pageNumber, pageSize, and optional query parameters
- iii. return `AutoRentalListDTO` containing contents of `List<AutoRentalDTO>`

- e. delete a specific resource

- f. delete all instances of the resource

Example Skeletal Controller Method

```
@RequestMapping(path=AutosAPI.AUTO_PATH,  
    method = RequestMethod.POST,  
    consumes = {...},  
    produces = {...})  
public ResponseEntity<AutoDTO> createAuto(@RequestBody AutoDTO newAuto) {  
    throw new RuntimeException("not implemented");  
    //or  
    return ResponseEntity.status(HttpStatus.CREATED).body(...);  
}
```

3. CLIENT TODO: Identify the response status codes to be returned for each of the actions
 - a. account for success and failure conditions
 - b. authorization does not need to be taken into account at this time

182.4. Grading

Your solution will be evaluated on:

1. identify resources
 - a. whether your identified resource(s) represent thing(s)
2. define a URI for a resource
 - a. whether the URI(s) center on the resource versus actions performed on the resource
3. define the proper method for a call against a resource
 - a. whether proper HTTP methods have been chosen to represent appropriate actions
4. CLIENT TODO: identify appropriate response code family and value to use in certain circumstances
 - a. whether proper response codes been identified for each action

182.5. Additional Details

1. This portion of the assignment alone should produce a `@RestController` class with annotated methods that statically define your API interface (possibly missing content details). There is nothing to run or test in this portion alone.
2. A simple and useful way of expressing your URIs can be through defining a set of public static attributes expressing the collection and individual instance of the resource type.

Example Template Resource Declaration

```
String (RESOURCE)S_PATH="(path)";  
String (RESOURCE)_PATH="(path)/{identifier(s)}";
```

3. If you start with this portion, you may find it helpful to
 - a. create sparsely populated DTO classes—`AutoRentalDTO` with just an `id` and `AutoRentalListDTO`—to represent the payloads that are accepted and returned from the methods
 - b. have the controller simply throw a `RuntimeException` indicating that the method is not yet implemented. That would be a good excuse to also establish an exception advice to handle thrown exceptions.
4. The details of the `AutoRental` will be performed server-side—based upon IDs and properties provided by the client and the `Auto` and `Renter` values found server-side. The client never provides more than an ID for an `Auto` or `Renter` and if it does—the server-side must ignore and rely on the server-side source for details.
5. There is nothing to code up relative to response codes at this point. However:
 - a. Finding zero resources to list is not a failure. It is a success with no resources in the collection.
 - b. Not finding a specific resource is a failure and the status code returned should reflect that.



Instances of Action Verbs can be Resource Nouns

If an action does not map cleanly to a resource+HTTP method, consider thinking of the action (e.g., `cancel`) as one instance of an action (e.g., `cancellation`) that is a sub-resource of the subject (e.g., `subjects/{subjectId}/cancellations`). How might you think of the action if it took days to complete? (e.g. a sub-resource with `POST/create()`, `GET/isComplete()`, `PUT/changePriority()`, and `DELETE/terminate()`)

Chapter 183. Assignment 2d: Client/API Interactions

183.1. Purpose

In this portion of the assignment, you will demonstrate your knowledge of designing and implementing the interaction between a Web client and API. You will:

1. implement a service method with Spring MVC synchronous annotated controller
2. implement a client using `RestTemplate` or `RestClient`
3. pass parameters between client and service over HTTP
4. return HTTP response details from service
5. access HTTP response details in client

183.2. Overview

In this portion of the assignment you will invoke your resource's Web API from a client running within a JUnit test case.

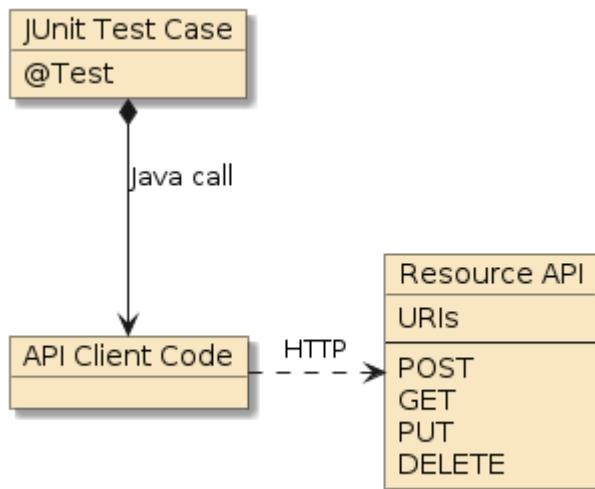


Figure 64. API/Service Interactions

There will be at least two primary tests in this portion of the assignment: handling success and handling failure. The failure will be either real or simulated through a temporary resource stub implementation.

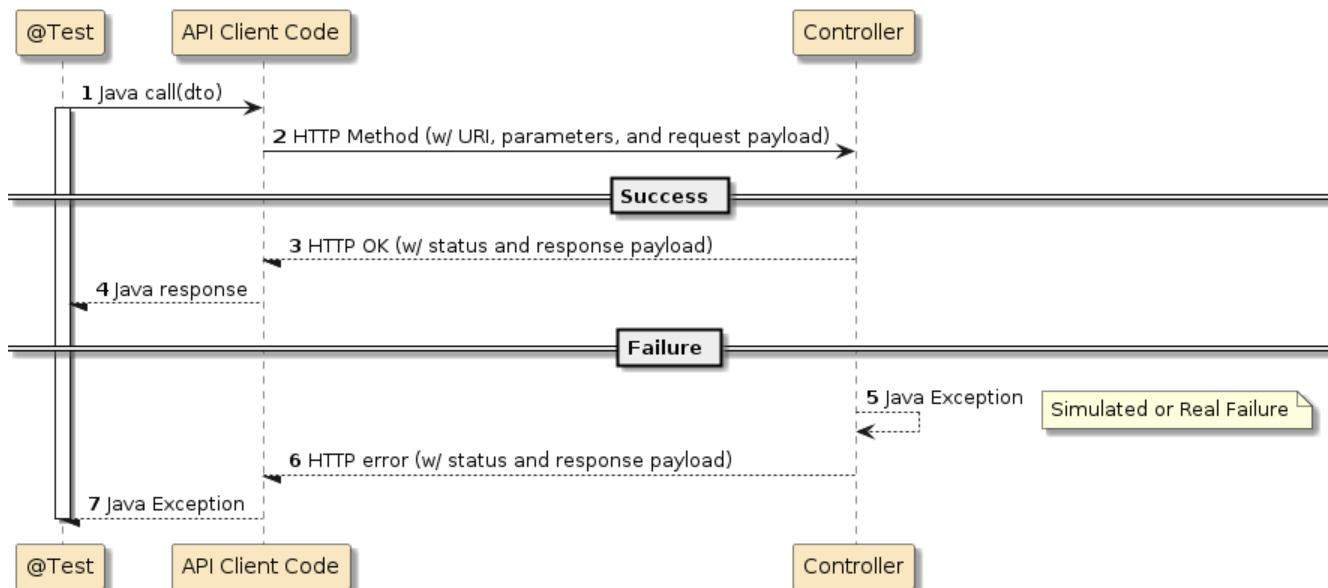


Figure 65. Two Primary Tests

183.3. Requirements

1. Implement stub behavior in the controller class as necessary to complete the example end-to-end calls.

Example Stub Response

```

AutoDTO newRental = AutoDTO.builder().id("1").build()
URI location = ServletUriComponentsBuilder.fromCurrentRequestUri()
    .replacePath(AUTORENTAL_PATH)
    .build("1");
return ResponseEntity.created(location).body(newRental);

```

2. Implement a unit integration test to demonstrate a success path

- a. use either a `RestTemplate` or `RestClient` API client class for this test. You may also leverage Spring HTTP Interface.
- b. make at least one call that passes parameter(s) to the service and the results of the call depend on that passed parameter value
- c. access the return status and payload in the JUnit test/client
- d. evaluate the result based on the provided parameter(s) and expected success status

Example Response Evaluation

```

then(response.getStatusCode()).isEqualTo(HttpStatus.CREATED);
then(response.getHeaders().getLocation()).isNotEmpty();
then(autoResult.getId()).isNotBlank();
then(autoResult).isEqualTo(autoRequestDTO.withId(autoResult.getId()));

```



Examples use `RestTemplate`

The Auto and Renter examples only use the `RestTemplate` and Spring HTTP Interface approaches.

One Success, One Failure, and Move On



Don't put too much work into more than a single success and failure path test before completing more of the end-to-end. Your status and details will likely change.

183.4. Grading

Your solution will be evaluated on:

1. implement a service method with Spring MVC synchronous annotated controller
 - a. whether your solution implements the intended round-trip behavior for an HTTP API call to a service component
2. implement a client using `RestTemplate` or `RestClient`
 - a. whether you are able to perform an API call using either the `RestTemplate`, `RestClient`, or Spring HTTP Interface APIs
3. pass parameters between client and service over HTTP
 - a. whether you are able to successfully pass necessary parameters between the client and service
4. return HTTP response details from service
 - a. whether you are able to return service response details to the API client
5. access HTTP response details in client
 - a. whether you are able to access HTTP status and response payload

183.5. Additional Details

1. The required end-to-end tests in the last section requires many specific success and failure test scenarios. View this portion of the assignment as just an early draft work of those scenarios. If you have completed the end-to-end tests, you have completed this portion of the assignment.
2. Your DTO class(es) have been placed in your Client module in a separate section of this assignment. You may want to add an **optional** API client class to that Client module—to encapsulate the details of the HTTP calls. The `autorenters-support-client` module contains example client API classes for Autos and Renters using `RestTemplate` and Spring HTTP Interface.
3. Avoid placing extensive business logic into the stub portion of the assignment. The controller method details are part of a separate section of this assignment.
4. This portion of the assignment alone should produce a simple, but significant demonstration of client/API communications (success and failure) using HTTP and service as the model for implementing additional resource actions.
5. Inject the dependencies for the test from the Spring context. Anything that depends on the server's port number must be delayed (`@Lazy`)

```
@Bean @Lazy ②
public ServerConfig serverConfig(@LocalServerPort int port) { ①
    return new ServerConfig().withPort(port).build();
}

@Bean @Lazy ③
public AutosAPI autosAPI(RestTemplate restTemplate, ServerConfig serverConfig) {
    return new AutosAPIClient(restTemplate, serverConfig, MediaType
.APPLICATION_JSON);
}

@SpringBootTest(...webEnvironment=...
public class AutoRentalAPITest {
    @Autowired
    private AutosAPI autoAPI;
```

① server's port# is not known until runtime

② cannot eagerly create `@Bean` until server port number available

③ cannot eagerly create dependents of port number

Chapter 184. Assignment 2e: Service/Controller Interface

184.1. Purpose

In this portion of the assignment, you will demonstrate your knowledge of separating the Web API facade details from the service implementation details and integrating the two. You will:

1. implement a service class to encapsulate business logic
2. turn @RestController class into a facade and delegate business logic details to an injected service class
3. implement an error reporting strategy

184.2. Overview

In this portion of the assignment you will be implementing the core of the AutoRental components and integrating them as seamlessly as possible.

- the controller will delegate commands to a service class to implement the business logic.
- the service will use internal logic and external services to implement the details of the business logic.
- the repository will provide storage for the service.

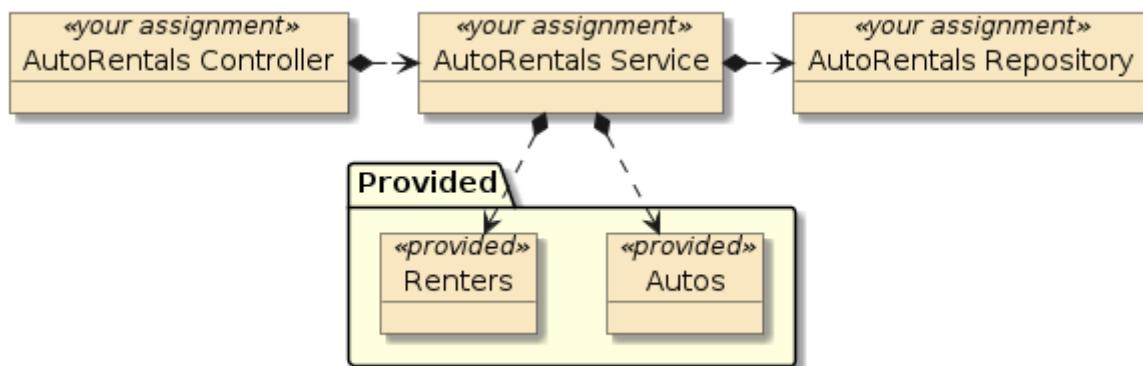


Figure 66. Service/Controller Interface

Your Assignment is Primarily AutoRental and Integration



You have been provided complete implementation of the **Auto** and **Renter** services. You only have to implement the **AutoRental** components and integration that with **Auto** and **Renter** services.

A significant detail in this portion of the assignment is to design a way to convey success and failure when carrying out an API command. The controller should act only as a web facade. The service(s) will implement the details of the services and report the results.

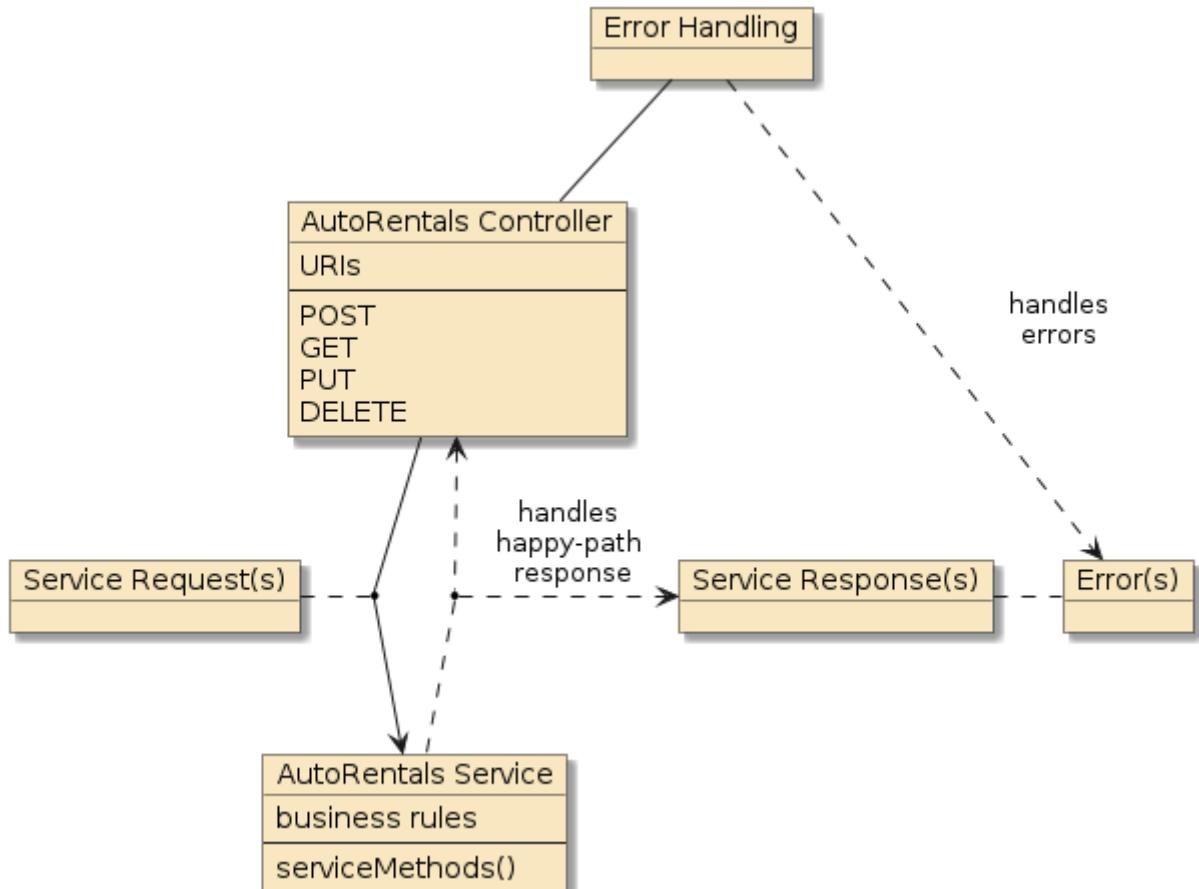


Figure 67. Service/Controller Interface

Under the hood of the `AutoRentalService` is a repository and external clients.

- You will create a `AutoRentalService` interface that uses `AutoRentalDTO` as its primary data type. This interface can be made reusable through the full semester of assignments.

This assignment will only work with DTO types (no entities/BOs) and a simulated/stub Repository.

- You will create a repository interface and implementation that mimic the behavior of a CRUD and Pageable Repository of a future assignment.
- You will inject and implement calls to the Auto and Renter services.

One Application

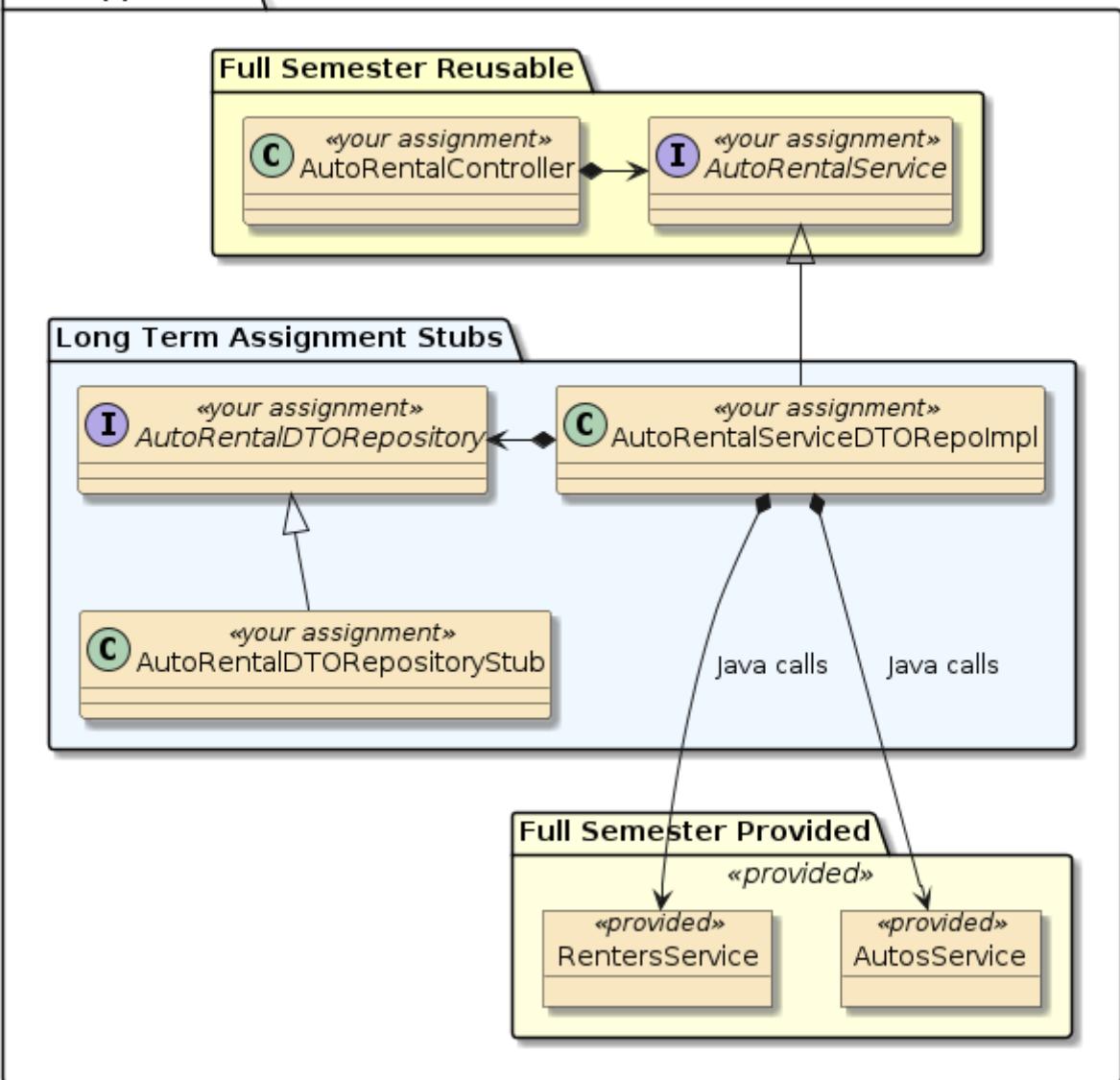


Figure 68. Assignment Components

184.3. Requirements

1. Implement a `AutoRentalDTORespository` interface and implementation component to simulate necessary behavior (e.g., `save`, `findById`, `find`) for the base `AutoRentalDTO` resource type. **Don't go overboard here.** We just need some place to generate IDs and hold the data in memory.
 - a. implement a Java interface (e.g., `AutoRentalDTORespository`).



Try to make this interface **conceptually** consistent with the Spring Data `ListCrudRepository` and `ListPagingAndSortingRepository` (including the use of `Pageable` and `Page`) to avoid changes later on. This is just a tip and not a requirement — implement what you need for now. Start with just `save()`.

- b. implement a component class stub (e.g., `AutoRentalDTORespositoryStub`) using simple, in-memory storage (e.g., `HashMap` or `ConcurrentHashMap`) and an ID generation mechanism (e.g., `int` or `AtomicInteger`)



You are free to make use of the `POJORepositoryMapImpl<T>` class in the `autorentals_support_api` module as your implementation for the repository. It comes with a `POJORepository<T>` interface and the Renter repository and service provide an example of its use. Report any bugs you find.

2. Implement a AutoRental service to implement actions and enforce business logic on the base resources
 - a. implement a Java interface This will accept and return AutoRentalDTO types.
 - b. implement a component class for the service.
 - c. inject the dependencies required to implement the business logic
 - i. (*provided*) `AutosService` - to verify existence of and obtain details of autos
 - ii. (*provided*) `RentersService` - to verify existence of and obtain details of renters
 - iii. (your assignment) `AutoRentalDTORepository` - to store details that are important to auto rentals



You are injecting the service implementations (not the HTTP API) for the Auto and Renter services into your AutoRental service. That means they will be part of your application and you will have a Java ⇒ Java interface with them.

- d. implement the business logic for the service
 - i. a AutoRental can only be created for an existing Auto and Renter. It will be populated using the values of that Auto and Renter on the server-side.
 - `autoId` (mandatory input) — used to locate the details of the Auto on the server-side
 - `renterId` (mandatory input) — used to locate the details of the Renter on the server-side
 - `startDate` (mandatory input) — used to indicate when the Rental will begin. This must be in the future and before or equal to `endDate`.
 - `endDate` (mandatory input) — used to indicate when the Rental will end. This must be after or equal to `beginDate`.
 - ii. AutoRental must populate the following fields from the Auto and Renter **obtained server-side** using the `autoId` and `renterId`.
 - from the server-side Auto
 - `makeModel` — derived from the Auto's `make` and `model` details obtained server-side
 - `amount` — calculated from `Auto.dailyRate * days in time span`
 - from the server-side Renter
 - `renterName` — derived from the Renter's `firstName` and `lastName` details on the server-side
 - `renterAge` — calculated from the `startDate` and Renter `dob` details on the server-side. A Renter must be at least 21 on `startDate` to be valid. The rejection shall

- include the text "too young".
- iii. `beginDate/endDate` time span cannot overlap with another AutoRental for the same Auto for a different Renter. The rejection shall include the text "conflict".
 - iv. a successful request to create a AutoRental will be returned with
 - the above checks made, id assigned, and a 201/CREATED http status returned
 - properties (`makeModel`, `renterName` `renterAge`, `amount`) filled in
 - v. `getAutoRental` returns current state of the requested AutoRental
 - vi. implement a paged `findAutoRentals` that returns all matches. Use the Spring Data `Pageable` and `Page` (and `PageImpl`) classes to express `pageNumber`, `pageSize`, and `page` results (i.e., `Page findAutoRentals(Pageable)`). You do not need to implement sort.
 - vii. augment the `findAutoRentals` to optionally include a search for matching `autoId`, `renterId`, time span (`startDate` and `endDate`), or any combination.

Implement Search Details within Repository class



Delegate the gory details of searching through the data—to the repository class.

3. Design a means for service calls to
 - a. indicate success
 - b. indicate failure to include internal or client error reason. Client error reasons must include separate issues "not found" and "bad request" at a minimum.
4. Integrate services into controller components
 - a. complete and report successful results to API client
 - b. report errors to API client, to include the status code and a textual message that is specific to the error that just occurred
5. Implement a unit integration test to demonstrate at least one success and error path
 - a. access the return status and payload in the client
 - b. evaluate the result based on the provided parameter(s) and expected success/failure status

184.4. Grading

Your solution will be evaluated on:

1. implement a service class to encapsulate business logic
 - a. whether your service class performs the actions of the service and acts as the primary enforcer of stated business rules
2. turn `@RestController` class into a facade and delegate business logic details to an injected service class
 - a. whether your API tier of classes act as a thin adapter facade between the HTTP protocol and service component interactions

3. implement an error reporting strategy
 - a. whether your design has identified how errors are reported by the service tier and below
 - b. whether your API tier is able to translate errors into meaningful error responses to the client

184.5. Additional Details

1. This portion of the assignment alone primarily provides an implementation pattern for how services will report successful and unsuccessful requests and how the API will turn that into a meaningful HTTP response that the client can access.
2. The `autorenters-support-api-svc` module contains a set of example DTO Repository Stubs.
 - The `Autos` package shows an example of a fully exploded implementation. Take this approach if you wish to write all the code yourself.
 - The `Renters` package shows an example of how to use the templated `POJORepository<T>` interface and `POJORepositoryMapImpl<T>` implementation. Take this approach if you want to delegate to an existing implementation and only provide the custom query methods.



`POJORepositoryMapImpl<T>` provides a protected `findAll(Predicate<T> predicate, Pageable pageable)` that returns a `Page<T>`. All you have to provide are the predicates for the custom query methods.

3. You are required to use the `Pageable` and `Page` classes (from the `org.springframework.data.domain` Java package) for paging methods in your finder methods — to be forward compatible with later assignments that make use of Spring Data. You can find example use of `Pageable` and `Page` (and `PageImpl`) in Auto and Renter examples.
4. It is highly recommend that exceptions be used between the service and controller layers to identify error scenarios and specific exceptions be used to help identify which kind of error is occurring in order to report accurate status to the client. Leave non-exception paths for successful results. The Autos and Renters example leverage the exceptions defined in the `ejava-dto-util` module. You are free to define your own.
5. It is highly recommended that `ExceptionHandlers` and `RestExceptionAdvice` be used to handle exceptions thrown and report status. The Autos and Renters example leverage the `ExceptionHandlers` from the `ejava-web-util` module. You are free to define your own.

Chapter 185. Assignment 2f: Required Test Scenarios

There are a set of minimum scenarios that are required of a complete project that must be specifically demonstrated in the submission.

1. Creation of AutoRental
 - a. success (201/**CREATED**)
 - b. failed creation because Auto unknown (422/**UNPROCESSABLE_ENTITY**)
 - c. failed creation because Auto unavailable for time span (422/**UNPROCESSABLE_ENTITY**)
 - d. failed creation because Renter unknown (422/**UNPROCESSABLE_ENTITY**)
 - e. failed creation because Renter too young (422/**UNPROCESSABLE_ENTITY**)
2. Update of AutoRental
 - a. success (200/**OK**)
 - b. failed because AutoRental does not exist (404/**NOT_FOUND**)
 - c. failed because AutoRental exists but change is invalid (422/**UNPROCESSABLE_ENTITY**)

185.1. Scenario: Creation of AutoRental

In this scenario, an AutoRental is attempted to be created.

185.1.1. Primary Path: Success

In this primary path, the Auto and Renter exist, all business rules are satisfied, and the API client is able to successfully create an AutoRental. The desired status in the response is a 201/CREATED. A follow-on query for AutoRentals will report the new entry.

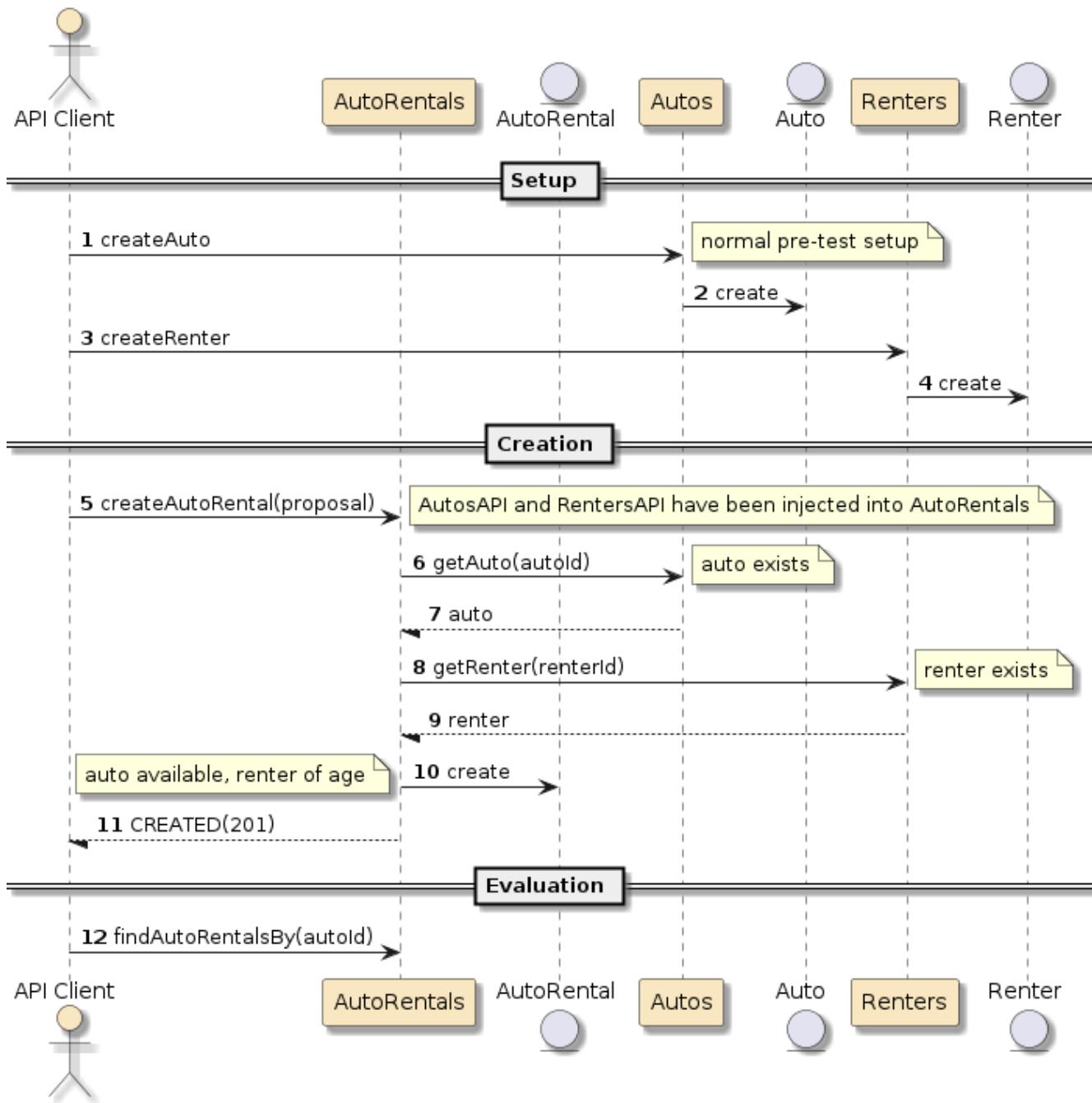


Figure 69. Creation of AutoRental for a Auto

185.1.2. Alternate Path: Auto unknown

In this alternate path, the Auto does not exist and the API client is unable to create a AutoRental. The desired response status is a 422/UNPROCESSABLE_ENTITY. The AutoRentals resource understood the request (i.e., not a 400/BAD_REQUEST or 404/NOT_FOUND), but request contained information that could not be processed.

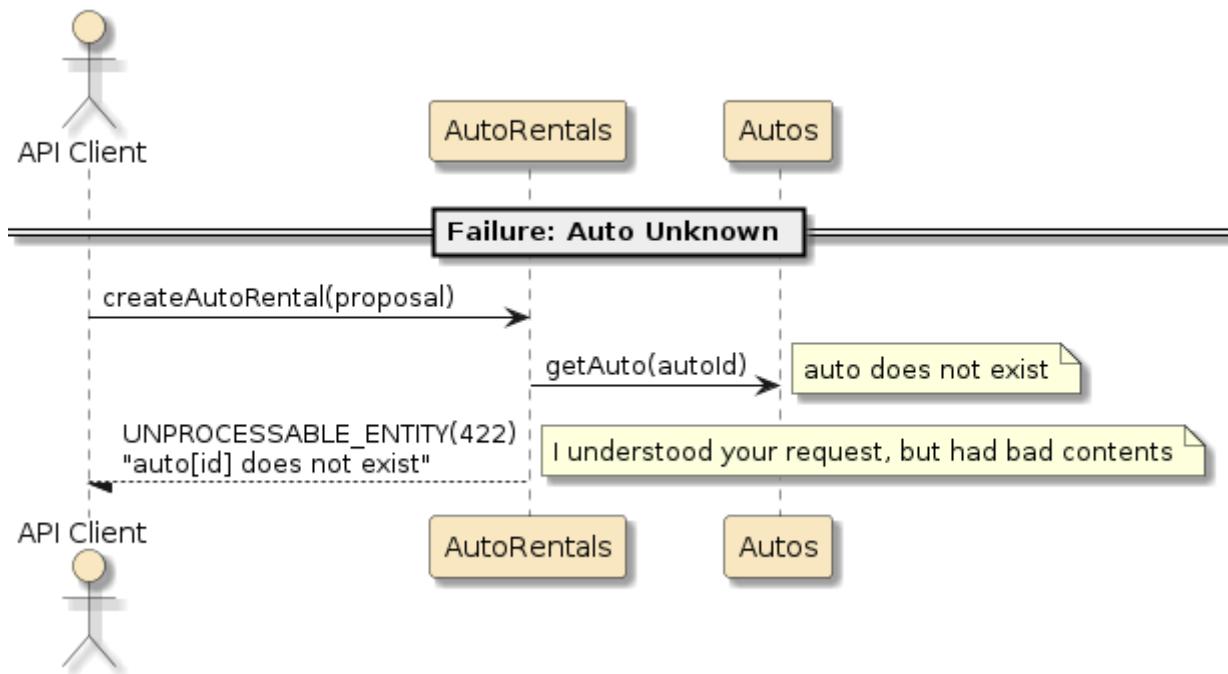


Figure 70. Auto unknown



`getAuto()` will return a 404/NOT_FOUND—which is not the same status requested here. AutoRentals will need to account for that difference.

185.1.3. Alternate Path: Auto Unavailable

In this alternate path, the Auto exists but is unavailable for the requested time span. The desired response status is a 422/UNPROCESSABLE_ENTITY. The AutoRentals resource understood the request, but request contained information that could not be processed.

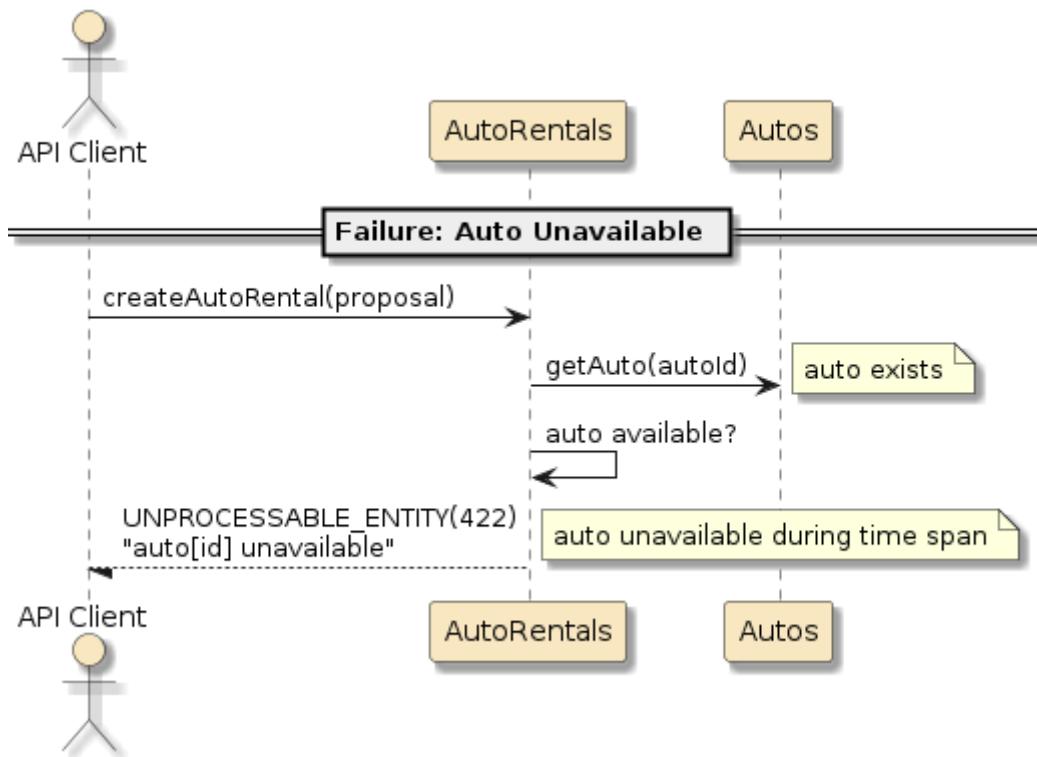


Figure 71. Auto unavailable

185.1.4. Alternate Path: Renter Unknown

In this alternate path, the Renter does not exist and the API client is unable to create a AutoRental for the Renter. The desired response status is a 422/UNPROCESSABLE_ENTITY.

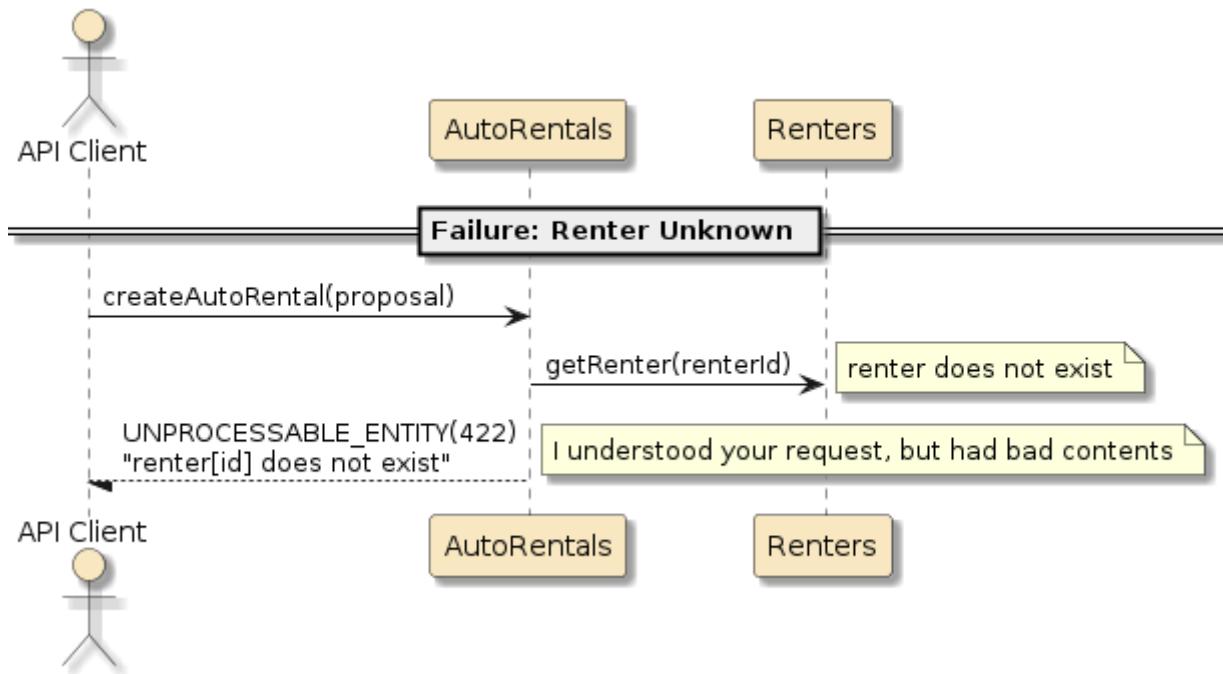


Figure 72. Renter unknown



`getRenter()` will return a 404/NOT_FOUND—which is not the same status requested here. AutoRentals will need to account for that difference.

185.1.5. Alternate Path: Renter Too Young

In this alternate path, the Renter exists but is too young to complete a AutoRental. The desired response status is a 422/UNPROCESSABLE_ENTITY.

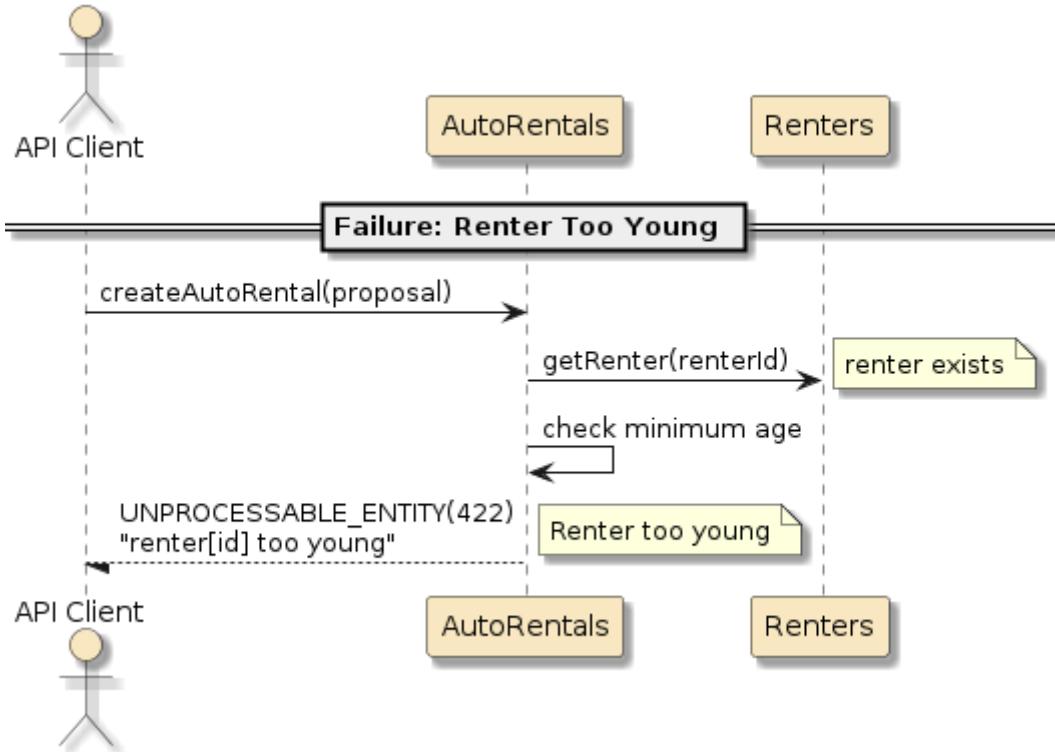


Figure 73. Renter too young

185.2. Scenario: Update of AutoRental

In this scenario, a AutoRental is attempted to be updated.

185.2.1. Primary Path: Success

In this primary path, the API client is able to successfully update the time span for an AutoRental. This update should be performed all on the server-side. The client primarily expresses an updated proposal and the business rules pass. A follow-on query for AutoRentals will report the updated entry.

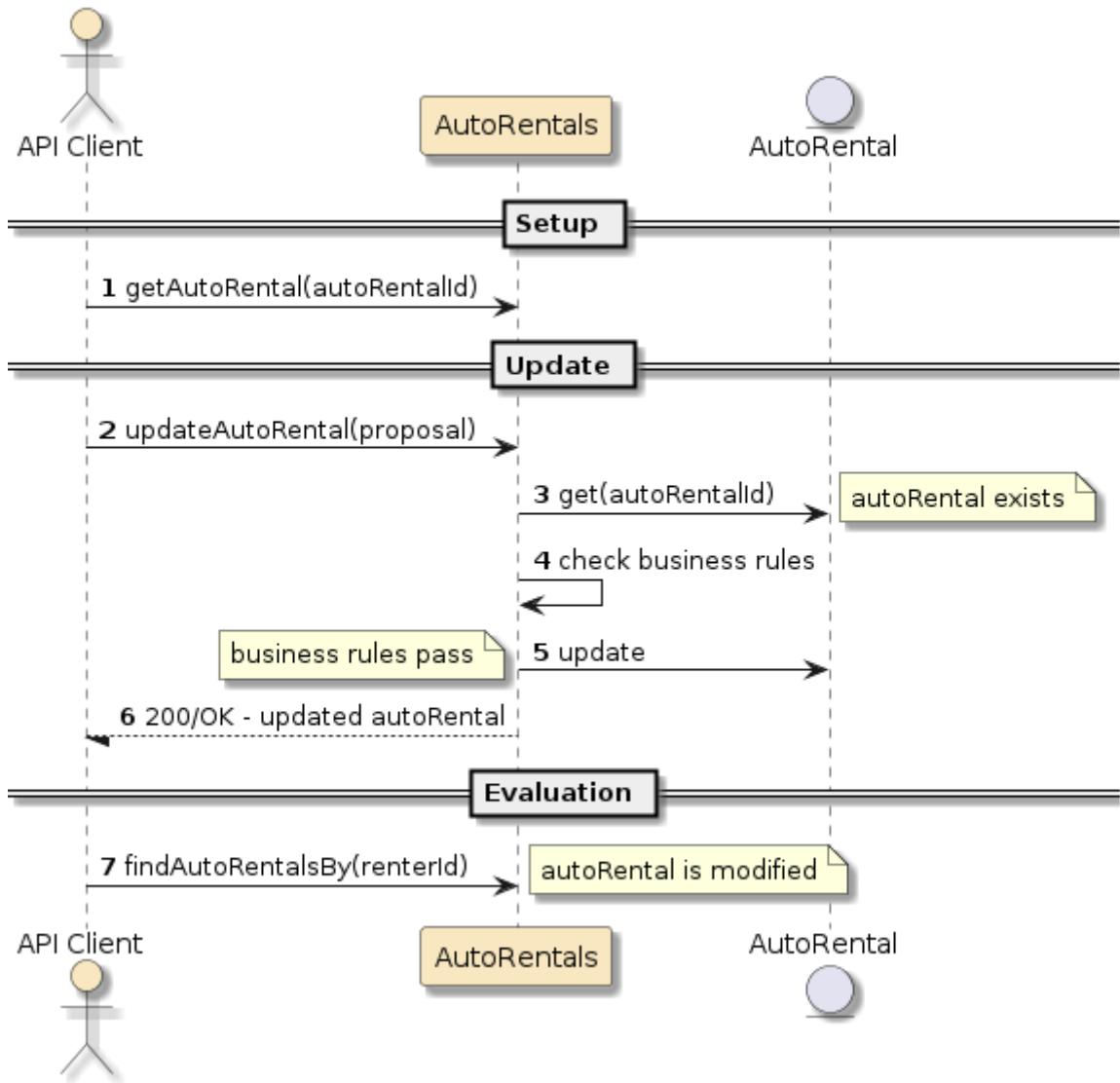


Figure 74. AutoRental update

185.2.2. Alternate Path: AutoRental does not exist

In this alternate path, the requested AutoRental does not exist. The expected response status code should be 404/NOT_FOUND to express that the target resource could not be found.

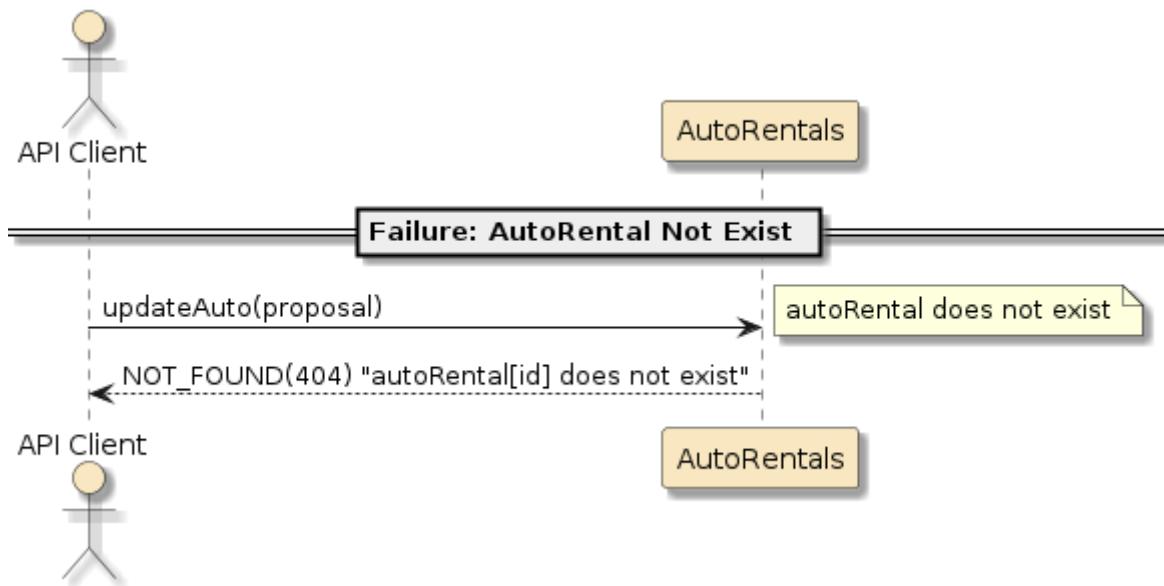


Figure 75. AutoRental does not exist

185.2.3. Alternate Path: Invalid Change

In this alternate path, the requested AutoRental exists but the existing Auto is not available for the new time span. The expected response status code should be 422/UNPROCESSABLE_ENTITY.

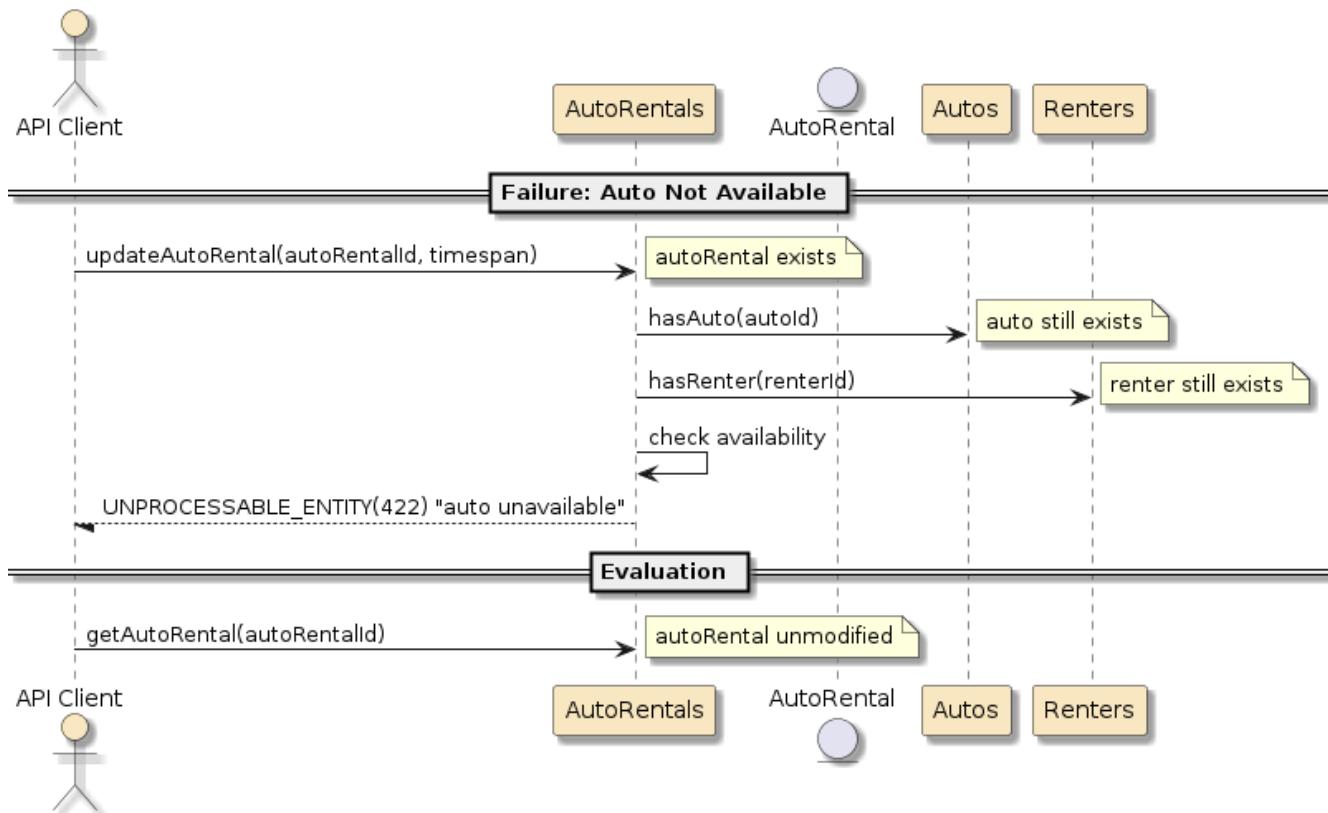


Figure 76. AutoRental invalid change

The scenario is showing that only the time span is being changed and the existing Auto and Renter should be used. The scenario is also showing that the existing Auto and Renter are being verified to still exist.



185.3. Requirements

1. Implement the above scenarios within one or more integration unit tests.
2. Name the tests such that they are picked up and executed by the Surefire test phase of the maven build.
3. Turn in a cleaned source tree of the project under a single root parent project. The Auto and Renter modules do not need to be included.
4. The source tree should be ready to build in an external area that has access to the ejava-nexus-snapshots repository.

185.4. Grading

1. create an integration test that verifies a successful scenario
 - a. whether you implemented a set of integration unit tests that verify the primary paths for AutoRentals
2. create an integration test that verifies a failure scenario
 - a. whether you implemented a set of integration unit tests that verify the failure paths for AutoRentals.

185.5. Additional Details

1. Place behavior in the proper place
 - a. The unit integration test is responsible for populating the Autos and Renters. It will supply AutoDTOs and RenterDTOs populated on the client-side—to the Autos and Renters APIs/services.
 - b. The unit integration test will pass sparsely populated AutoRentalDTOs to the server-side with autoId, renterId, etc. values express inputs for creating a listing or making a purchase. All details to populate the returned **AutoRentalDTOs** (i.e., Auto and Renter info) will come from the server-side. There should never be a need for the client to self-create/fully-populate a AutoRentalDTO.

Spring Security Introduction

copyright © 2024 jim stafford (jim.stafford@jhu.edu)

Chapter 186. Introduction

Much of what we have covered to date has been focused on delivering functional capability. Before we go much further into filling in the backend parts of our application or making deployments, we need to begin factoring in security concerns. Information Security is a practice of protecting information by mitigating risks ^[1]. Risks are identified with their impact and appropriate mitigations.

We won't get into the details of Information Security analysis and making specific trade-offs, but we will cover how we can address the potential mitigations through the use of a framework and how that is performed within Spring Security and Spring Boot.

186.1. Goals

You will learn:

- key terms relative to implementing access control and privacy
- the flexibility and power of implementing a filter-based processing architecture
- the purpose of the core Spring Authentication components
- how to enable Spring Security
- to identify key aspects of the default Spring Security

186.2. Objectives

At the conclusion of this lecture and related exercises, you will be able to:

1. define identity, authentication, and authorization and how they can help protect our software system
2. identify the purpose for and differences between encoding, encryption, and cryptographic hashes
3. identify the purpose of a filter-based processing architecture
4. identify the core components within Spring Authentication
5. identify where the current user authentication is held/located
6. activate default Spring Security configuration
7. identify and demonstrate the security features of the default Spring Security configuration
8. step through a series of calls through the Security filter chain

[1] "Information Security", Wikipedia

Chapter 187. Access Control

Access Control is one of the key mitigation factors within a security solution.

Identity

We need to know who the caller is and/or who is the request being made for. When you make a request in everyday life (e.g., make a pizza order)—you commonly have to supply your identity so that your request can be associated with you. There can be many layers of systems/software between the human and the action performed, so identity can be more complex than just a single value—but I will keep the examples to a simple username.

Authentication

We need verification of the requester's identity. This is commonly something known—e.g., a password, PIN, or generated token. Additional or alternate types of authentication like something someone has (e.g., access to a specific mobile phone number or email account, or assigned token generator) are also becoming more common today and are adding a needed additional level of security to more sensitive information.

Authorization

Once we know and can confirm the identity of the requester, we then need to know what actions they are allowed to perform and information they are allowed to access or manipulate. This can be based on assigned roles (e.g., administrator, user), relative role (e.g., creator, owner, member), or releasability (e.g., access markings).

These access control decisions are largely independent of the business logic and can be delegated to the framework. That makes it much easier to develop and test business logic outside the security protections and to be able to develop and leverage mature and potentially certified access control solutions.

Chapter 188. Privacy

Privacy is a general term applied to keeping certain information or interactions secret from others. We use various encoding, encryption, and hash functions in order to achieve these goals.

188.1. Encoding

Encoding converts source information into an alternate form that is safe for communication and/or storage.^[1] Two primary examples are [URL](#) and [Base64](#) encoding of special characters or entire values. Encoding may obfuscate the data, but by itself is not encryption. Anyone knowing the encoding scheme can decode an encoded value and that is its intended purpose.

Example Base64 encoding

```
$ echo -n jim:password | base64 ①
amlt0nBhc3N3b3Jk
$ echo -n amlt0nBhc3N3b3Jk | base64 -d
jim:password
```

① `echo -n` echos the supplied string without new line character added - which would pollute the value

188.2. Encryption

Encryption is a technique of encoding "plaintext" information into an enciphered form ("ciphertext") with the intention that only authorized parties—in possession of the encryption/decryption keys—can convert back to plaintext.^[2] Others not in possession of the keys would be forced to try to break the code through (hopefully) a significant amount of computation.

There are two primary types of keys—symmetric and asymmetric. For encryption with symmetric keys, the encryptor and decryptor must be in possession of the same/shared key. For encryption with asymmetric keys—there are two keys: public and private. Plaintext encrypted with the shared, public key can only be decrypted with the private key. SSH is an example of using asymmetric encryption.

Asymmetric encryption is more computationally intensive than symmetric



Asymmetric encryption is more computationally intensive than symmetric—so you may find that asymmetric encryption techniques will embed a dynamically generated symmetric key used to encrypt a majority of the payload within a smaller area of the payload that is encrypted with the asymmetric key.

Example AES Symmetric Encryption/Decryption

```
$ echo -n "jim:password" > /tmp/plaintext.txt ①
$ openssl enc -aes-256-cbc -salt -in /tmp/plaintext.txt -base64 \②
-pass pass:password > /tmp/ciphertext ③ ④
```

```

$ cat /tmp/ciphertext
U2FsdGVkX18mM2yNc337MS5r/iRJKI+roqkSym0zgMc=

$ openssl enc -d -aes-256-cbc -in /tmp/ciphertext -base64 -pass pass:password ⑤
jim:password

$ openssl enc -d -aes-256-cbc -in /tmp/ciphertext -base64 -pass pass:password123 ⑥
bad decrypt
4611337836:error:06FFF064:digital envelope routines:CRYPTO_internal:bad decrypt

```

- ① the payload we will be protecting is "jim:password"
- ② encrypting file of plaintext with a symmetric/shared key ("password"). Result is base64 encoded.
- ③ the symmetric encryption key is "password"
- ④ "-pass" parameter supplies password and "pass:" prefix identifies password is inline in the command
- ⑤ decrypting file of ciphertext with valid symmetric/shared key after being base64 decoded
- ⑥ failing to decrypt file of ciphertext with invalid key

188.3. Cryptographic Hash

A Cryptographic Hash is a one-way algorithm that takes a payload of an arbitrary size and computes a value of a known size that is unique to the input payload. The output is deterministic such that multiple, separate invocations can determine if they were working with the same input value—even if the resulting hash is not technically the same. Cryptographic hashes are good for determining whether information has been tampered with or to avoid storing recoverable password values.

Example MD5 Cryptographic Hash without Salt

```

$ echo -n password | md5
5f4dcc3b5aa765d61d8327deb882cf99 ①
$ echo -n password | md5
5f4dcc3b5aa765d61d8327deb882cf99 ①
$ echo -n password123 | md5
482c811da5d5b4bc6d497ffa98491e38 ②

```

- ① Core hash algorithms produce identical results for same inputs
- ② Different value produced for different input

Unlike encryption there is no way to mathematically obtain the original plaintext from the resulting hash. That makes it a great alternative to storing plaintext or encrypted passwords. However, there are still some unwanted vulnerabilities by having the calculated value be the same each time.

By adding some non-private variants to each invocation (called "Salt"), the resulting values can be technically different—making it difficult to use brute force [dictionary attacks](#). The following example uses the Apache `htpasswd` command to generate a Cryptographic Hash with a Salt value

that will be different each time. The first example uses the MD5 algorithm again and the second example uses the Bcrypt algorithm—which is more secure and widely accepted for creating Cryptographic Hashes for passwords.

Example MD5 Cryptographic Hash with Salt

```
$ htpasswd -bnm jim password  
jim:$apr1$ctN0ftbV$ZHs/IA3yt0jx0IZEZ1w5. ①  
  
$ htpasswd -bnm jim password  
jim:$apr1$gLU9VlAl$ihD0zr8PdiCRjF3pna2EE1 ①  
  
$ htpasswd -bnm jim password123  
jim:$apr1$9sJN0ggs$xvqrmNXLq0XZWjMSN/WLG.
```

① Salt added to help defeat dictionary lookups

Example Bcrypt Cryptographic Hash with Salt

```
$ htpasswd -bnBC 10 jim password  
jim:$2y$10$cBJ0zUbDurA32SOSC.AnEuhUW269ACaPM7tDtD9vbrEg14i9GdGaS  
  
$ htpasswd -bnBC 10 jim password  
jim:$2y$10$RztUum5dBjKrcgiBNQlTHueqDFd60RByYgQPbugPCjv23V/RzfdVG  
  
$ htpasswd -bnBC 10 jim password123  
jim:$2y$10$s0I8X22Z1k2wK43S7dUBjup2VI1WUaJwfzX8Mg2Ng0jBxnjCEA0F2
```

htpasswd Command Line

htpasswd [options] [username] [password]

- -b: accept password from command line versus prompting for it
- -n: display results to stdout
- -B: use Bcrypt encryption
- -m: use md5 encryption
- -C: set computing time between 4 and 17; high is slower and more secure



[1] "Code-Encoding", Wikipedia

[2] "Encryption", Wikipedia

Chapter 189. Spring Web

Spring Framework operates on a series of core abstractions and a means to leverage them from different callchains. Most of the components are manually assembled through builders and components/beans are often integrated together through the Spring application context.

For the web specifically, the callchains are implemented through an initial web interface implemented through the hosting or embedded web server. Often the web.xml will define a certain set of filters that add functionality to the request/response flow.

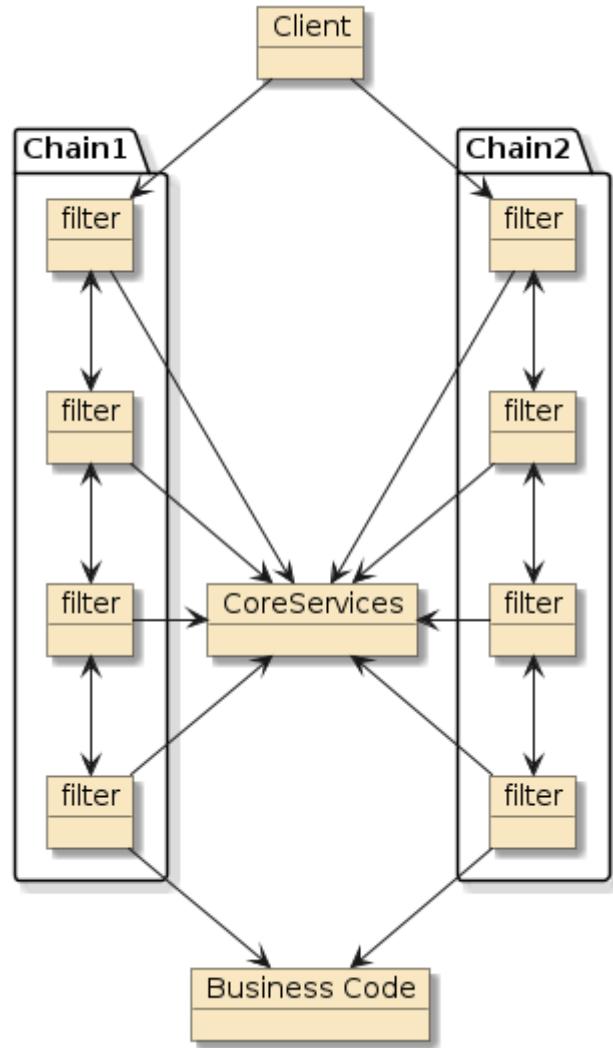


Figure 77. Spring Web Framework Operates through Flexibly Assembled Filters and Core Services

Chapter 190. No Security

We know by now that we can exercise the Spring Application Filter Chain by implementing and calling a controller class. I have implemented a simple example class that I will be using throughout this lecture. At this point in time — no security has been enabled.

190.1. Sample GET

The example controller has two example GET calls that are functionally identical at this point because we have no security enabled. The following is registered to the `/api/anonymous/hello` URI and the other to `/api/authn/hello`.

Example GET

```
@RequestMapping(path="/api/anonymous/hello",
    method= RequestMethod.GET)
public String getHello(@RequestParam(name = "name", defaultValue = "you") String name)
{
    return "hello, " + name;
}
```

We can call the endpoint using the following curl or equivalent browser call.

Calling Example GET

```
$ curl -v -X GET "http://localhost:8080/api/anonymous/hello?name=jim"
> GET /api/anonymous/hello?name=jim HTTP/1.1
< HTTP/1.1 200
< Content-Length: 10
<
hello, jim
```

190.2. Sample POST

The example controller has three example POST calls that are functionally identical at this point because we have no security or other policies enabled. The following is registered to the `/api/anonymous/hello` URI. The other two are mapped to the `/api/authn/hello` and `/api/alt/hello` URIs.^[1].

Example POST

```
@RequestMapping(path="/api/anonymous/hello",
    method = RequestMethod.POST,
    consumes = MediaType.TEXT_PLAIN_VALUE,
    produces = MediaType.TEXT_PLAIN_VALUE)
public String postHello(@RequestBody String name) {
    return "hello, " + name;
```

```
}
```

We can call the endpoint using the following curl command.

Calling Example POST

```
$ curl -v -X POST "http://localhost:8080/api/anonymous/hello" \
-H "Content-Type: text/plain" -d "jim"
> POST /api/anonymous/hello HTTP/1.1
< HTTP/1.1 200
< Content-Length: 10
<
hello, jim
```

190.3. Sample Static Content

I have not mentioned it before now—but not everything served up by the application has to be live content provided through a controller. We can place static resources in the `src/main/resources/static` folder and have that packaged up and served through URIs relative to the root.

static resource locations

Spring Boot will serve up static content found in `/static`, `/public`, `/resources`, or `/META-INF/resources/` of the classpath.

 `src/main/resources/
`-- static
 '-- content
 '-- hello_static.txt`

Anything placed below `src/main/resources` will be made available in the classpath within the JAR via `target/classes`.

`target/classes/
`-- static <== classpath:/static at runtime
 '-- content <== /content URI at runtime
 '-- hello_static.txt`

This would be a common thing to do for css files, images, and other supporting web content. The following is a text file in our sample application.

`src/main/resources/static/content/hello_static.txt`

`Hello, static file`

The following is an example GET of that static resource file.

```
$ curl -v -X GET "http://localhost:8080/content/hello_static.txt"
> GET /content/hello_static.txt HTTP/1.1
< HTTP/1.1 200
< Content-Length: 19
<
Hello, static file
```

Chapter 191. Spring Security

The [Spring Security](#) framework is integrated into the web callchain using filters that form an internal Security Filter Chain.

We will look at the Security Filter Chain in more detail shortly. At this point—just know that the framework is a flexible, filter-based framework where many different authentication schemes can be enabled. Let's take a look first at the core services used by the Security Filter Chain.

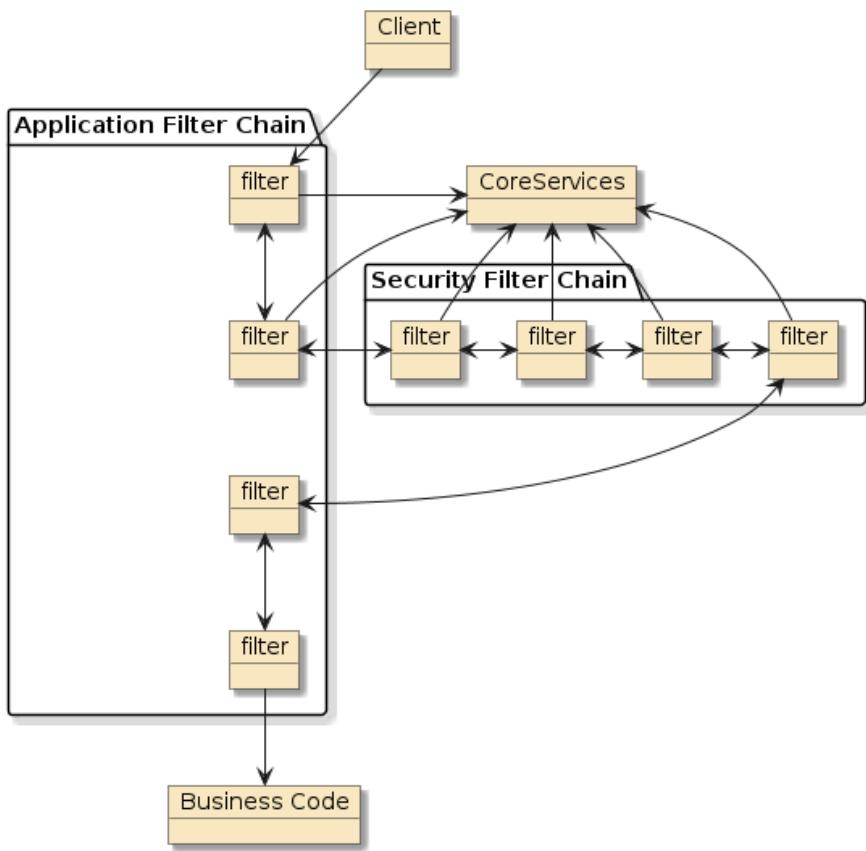


Figure 78. Spring Security Implemented as Extension of Application Filter Chain

191.1. Spring Core Authentication Framework

Once we enable Spring Security—a set of core authentication services are instantiated and made available to the Security Filter Chain. The key players are a set of interfaces with the following roles.

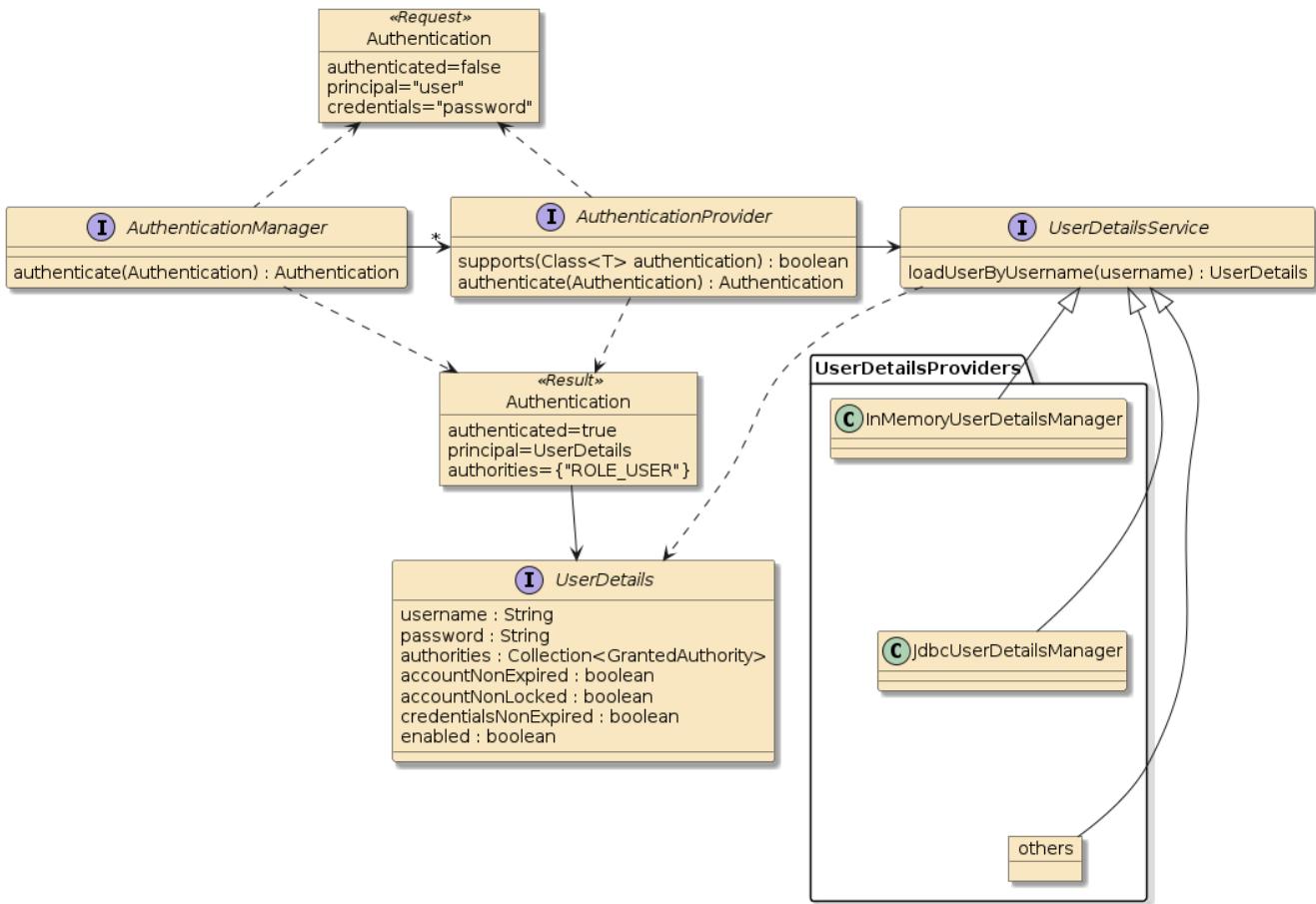


Figure 79. Spring Security Core Authentication Framework

Authentication

provides both an authentication request and result abstraction. All the key properties (principal, credentials, details) are defined as `java.lang.Object` to allow just about any identity and authentication abstraction be represented. For example, an **Authentication** request has a principal set to the username String and an **Authentication** response has the principal set to **UserDetails** containing the username and other account information.

AuthenticationManager

provides a front door to authentication requests that may be satisfied using one or more **AuthenticationProvider**

AuthenticationProvider

a specific authenticator with access to **UserDetails** to complete the authentication. **Authentication** requests are of a specific type. If this provider supports the type and can verify the identity claim of the caller—an **Authentication** result with additional user details is returned.

UserDetailsService

a lookup strategy used by **AuthenticationProvider** to obtain **UserDetails** by username. There are a few configurable implementations provided by Spring (e.g., JDBC), but we are encouraged to create our own implementations if we have a credentials repository that was not addressed.

UserDetails

an interface that represents the minimal needed information for a user. This will be made part

of the `Authentication` response in the `principal` property.

191.2. SecurityContext

The authentication is maintained inside a `SecurityContext` that can be manipulated over time. The current state of authentication is located through static methods of the `SecurityContextHolder` class. Although there are multiple strategies for maintaining the current `SecurityContext` with `Authentication` — the most common is `ThreadLocal`.

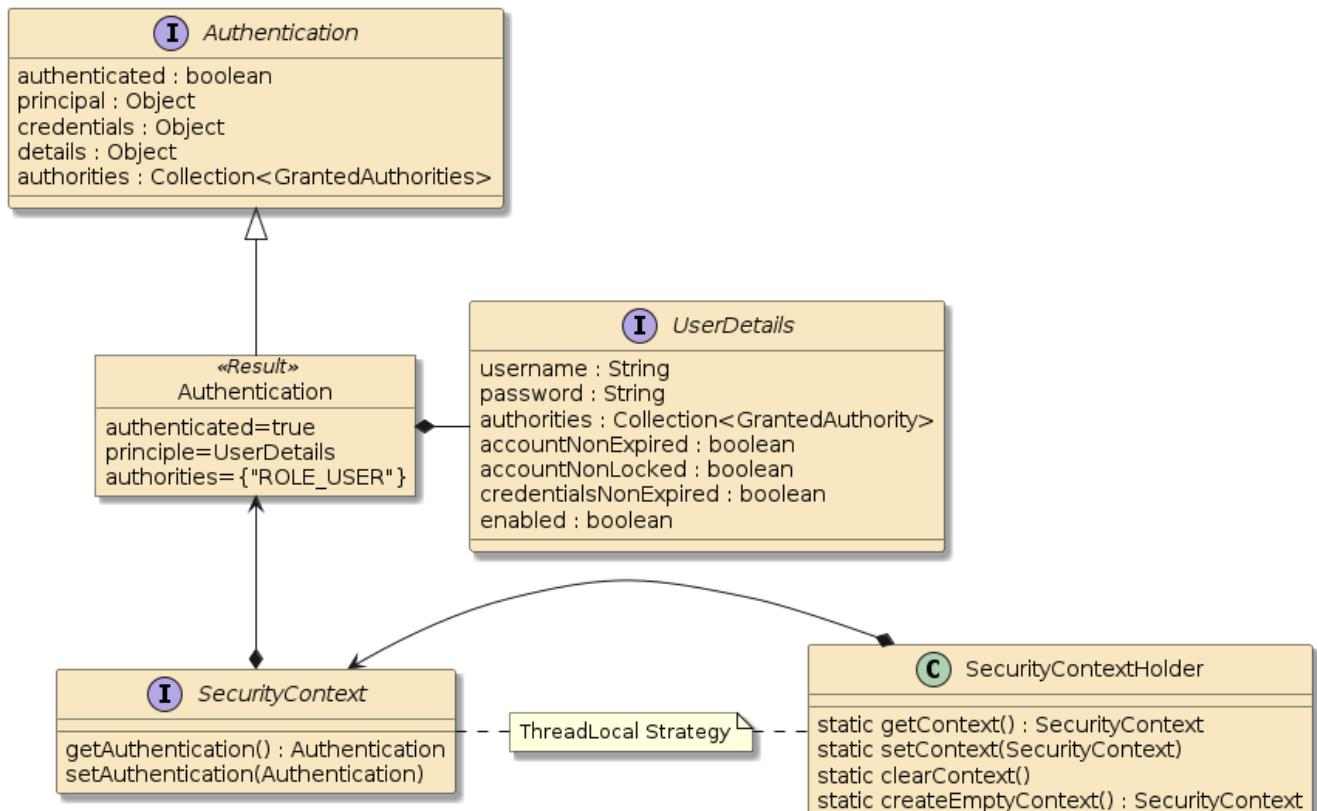


Figure 80. Current `SecurityContext` with `Authentication` accessible through `SecurityContextHolder`

Chapter 192. Spring Boot Security AutoConfiguration

As with most Spring Boot libraries — we have to do very little to get started. Most of what you were shown above is instantiated with a single additional dependency on the `spring-boot-starter-security` artifact.

192.1. Maven Dependency

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

This artifact triggers three (3) AutoConfiguration classes in the `spring-boot-autoconfiguration` artifact.

- For Spring Boot < 2.7, the autoconfiguration classes will be named in `META-INF/spring.factories`:

Spring Boot Starter Security (<= 2.7)

```
# org.springframework-boot:spring-boot-autoconfigure/META-INF/spring.factories
...
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
...
org.springframework.boot.autoconfigure.security.servlet.SecurityAutoConfiguration,\n
org.springframework.boot.autoconfigure.security.servlet.UserDetailsServiceAutoConfigur
ation,\n
org.springframework.boot.autoconfigure.security.servlet.SecurityFilterAutoConfiguratio
n,\n
```

- For Spring Boot >= 2.7, the autoconfiguration classes will be named in `META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports`

Spring Boot Starter Security (>= 2.7)

```
# org.springframework-boot:spring-boot-autoconfigure/META-
INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports
...
org.springframework.boot.autoconfigure.security.servlet.SecurityAutoConfiguration
org.springframework.boot.autoconfigure.security.servlet.UserDetailsServiceAutoConfigur
ation
org.springframework.boot.autoconfigure.security.servlet.SecurityFilterAutoConfiguratio
n
```

The details of this may not be that important except to understand how the default behavior was

assembled and how future customizations override this behavior.

192.2. SecurityAutoConfiguration

The `SecurityAutoConfiguration` imports two `@Configuration` classes that conditionally wire up the security framework discussed with default implementations.

- `SpringBootWebSecurityConfiguration` makes sure there is at least a default `SecurityFilterChain` (more on that later) which
 - requires all URIs be authenticated
 - activates FORM and BASIC authentication
 - enables CSRF and other security protections

Default Security Chain Configuration

```
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.web.SecurityFilterChain;
import static org.springframework.security.config.Customizer.withDefaults;

@Bean
@Order(SecurityProperties.BASIC_AUTH_ORDER) //very low priority
SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws
Exception {
    http.authorizeHttpRequests((requests) -> requests.anyRequest().
authenticated());
    http.formLogin(withDefaults());
    http.httpBasic(withDefaults());
    return http.build();
}
```

- `WebSecurityEnablerConfiguration` activates all security components by supplying the `@EnableWebSecurity` annotation when the security classes are present in the classpath.

Default Security Enabler

```
@Configuration(proxyBeanMethods = false)
@ConditionalOnMissingBean(name = BeanIds.SPRING_SECURITY_FILTER_CHAIN) ②
@ConditionalOnClass(EnableWebSecurity.class) ①
@EnableWebSecurity
static class WebSecurityEnablerConfiguration {
```

① activates when Spring Security is present

② deactivates when `springSecurityFilterChain` bean is defined

192.3. WebSecurityConfiguration

`WebSecurityConfiguration` is imported by `@EnableWebSecurity`. It gathers all the `SecurityFilterChain` beans for follow-on initialization into runtime `FilterChains`.

192.4. UserDetailsServiceAutoConfiguration

The `UserDetailsServiceAutoConfiguration` simply defines an in-memory `UserDetailsService` if one is not yet present. This is one of the provided implementations mentioned earlier—but still just a demonstration toy. The `UserDetailsService` is populated with one user:

- name: `user`, unless defined
- password: generated, unless defined

Example Output from Generated Password

```
Using generated security password: ff40aec-44c2-495a-bbbf-3e0751568de3
```

Overrides can be supplied in properties

Example Default user/password Override

```
spring.security.user.name: user
spring.security.user.password: password
```

192.5. SecurityFilterAutoConfiguration

The `SecurityFilterAutoConfiguration` establishes the `springSecurityFilterChain` filter chain, implemented as a `DelegatingFilterProxy`. The delegate of this proxy is supplied by the details of the `SecurityAutoConfiguration`.

Chapter 193. Default FilterChain

When we activated Spring security we added a level of filters to the Application Filter Chain. The first was a `DelegatingFilterProxy` that lazily instantiated the filter using a delegate obtained from the Spring application context. This delegate ends up being a `FilterChainProxy` which has a prioritized list of `SecurityFilterChain` (implemented using `DefaultSecurityFilterChain`). Each `SecurityFilterChain` has a `requestMatcher` and a set of zero or more `Filters`. Zero filters essentially bypasses security for a particular URI pattern.

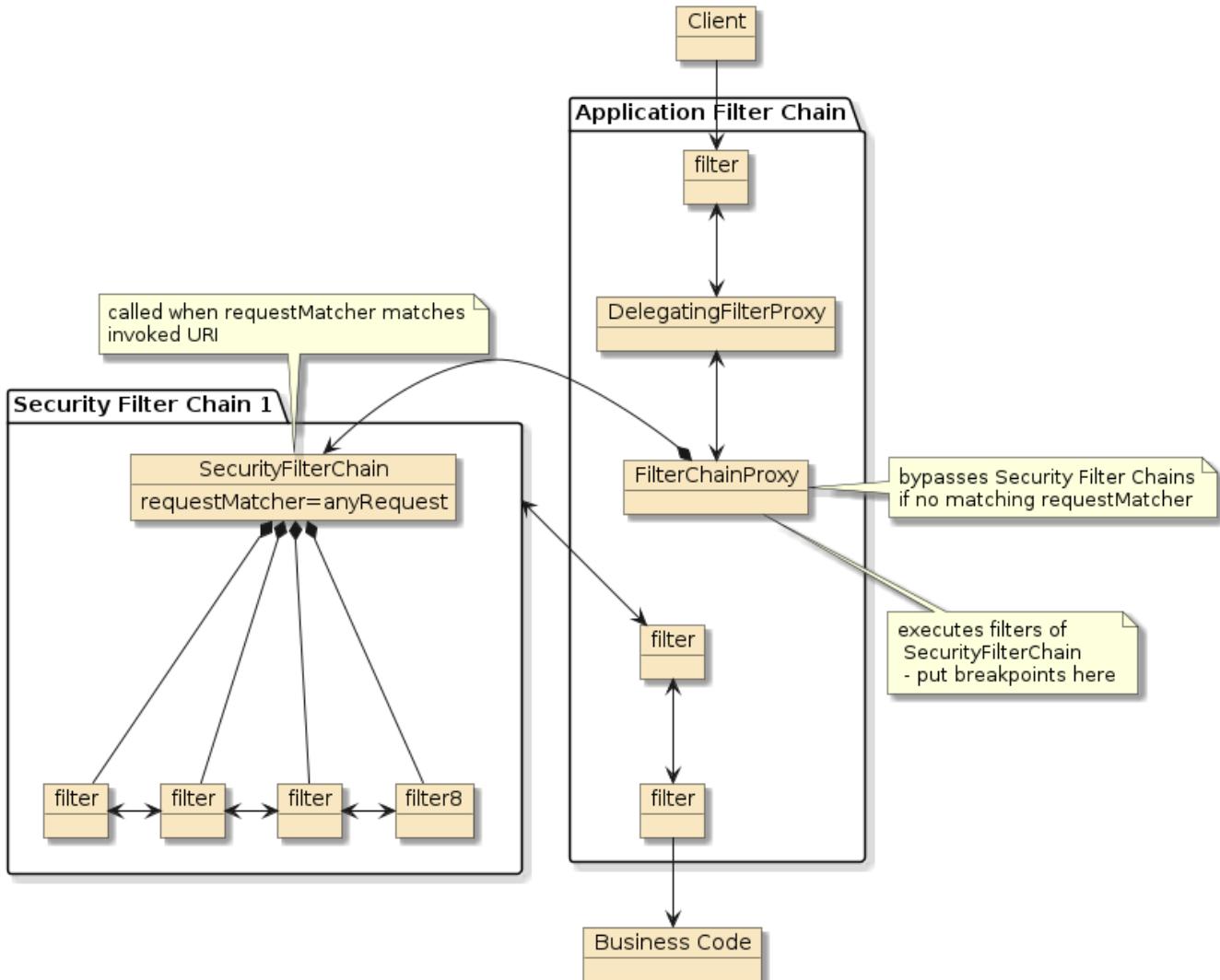


Figure 81. Default Security Filter Chain

Chapter 194. Default Secured Application

With all that said—and all we really did was add an artifact dependency to the project—the following shows where the AutoConfiguration left our application.

194.1. Form Authentication Activated

Form Authentication has been activated, and we are now stopped from accessing all URLs without first entering a valid username and password. Remember, the default username is `user` and the default password was output to the console unless we supplied one in properties. The following shows the result of a redirect when attempting to access any URL in the application.

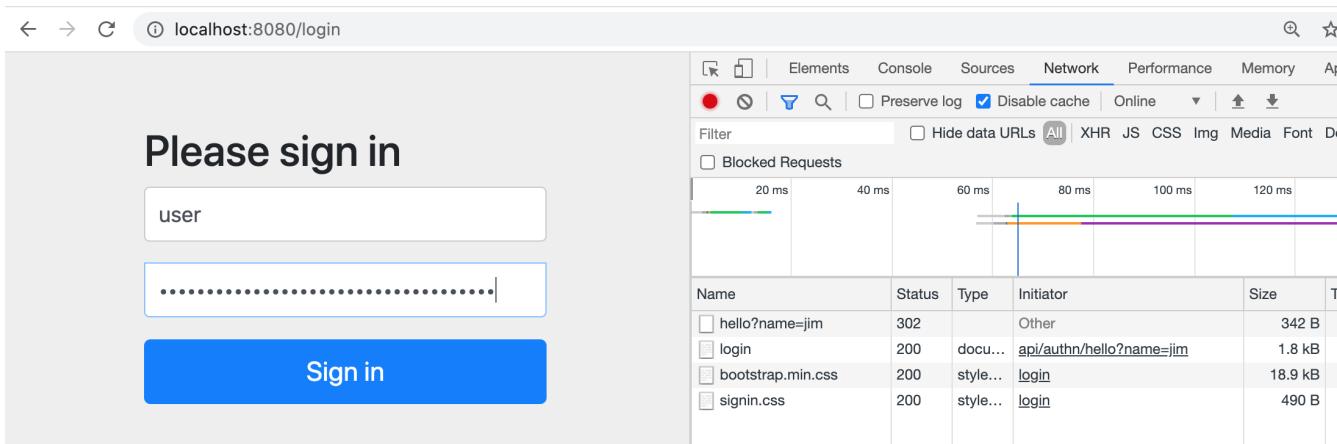


Figure 82. Example Default Form Login Activated

1. We entered <http://localhost:8080/api/anonymous/hello?name=jim>
2. Application saw there was no authentication for the session and redirected to /login page
3. Login URL, html, and CSS supplied by spring-boot-starter-security

If we call the endpoint from curl, without indicating we can visit an HTML page, we get flatly rejected with a 401/UNAUTHORIZED. The response does inform us that BASIC Authentication is available.

Example 401/Unauthorized from Default Secured Application

```
$ curl -v http://localhost:8080/api/authn/hello?name=jim
> GET /authn/hello?name=jim HTTP/1.1
< HTTP/1.1 401
< Set-Cookie: JSESSIONID=D124368C884557286BF59F70888C0D39; Path=/; HttpOnly
< WWW-Authenticate: Basic realm="Realm" ①
>{"timestamp":"2020-07-01T23:32:39.909+00:00","status":401,
"error":"Unauthorized","message":"Unauthorized","path":"/authn/hello"}
```

① `WWW-Authenticate` header indicates that BASIC Authentication is available

If we add an Accept header to the curl request with `text/html`, we get a 302/REDIRECT to the login page the browser automatically took us to.

Example 302/Redirect to FORM Login Page

```
$ curl -v http://localhost:8080/api/authn/hello?name=jim \
-H "Accept: text/plain,text/html" ①
> GET /authn/hello?name=jim HTTP/1.1
> Accept: text/plain, text/html
< HTTP/1.1 302
< Set-Cookie: JSESSIONID=E132523FE23FA8D18B94E3D55820DF13; Path=/; HttpOnly
< Location: http://localhost:8080/login
< Content-Length: 0
```

① adding an Accept header accepting text initiates a redirect to login form

The login (URI [/login](#)) and logout (URI [/logout](#)) forms are supplied as defaults. If we use the returned JSESSIONID when accessing and successfully completing the login form—we will continue on to our originally requested URL.

Since we are targeting APIs—we will be disabling that very soon and relying on more stateless authentication mechanisms.

194.2. Basic Authentication Activated

BASIC authentication is also activated by default. This is usable by our API out of the gate, so we will use this in examples. The following shows an example of BASIC encoding the `username:password` values in a Base64 string and then supplying the result of that encoding in an `Authorization` header prefixed with the word "BASIC".

Example Successful Basic Authentication

```
$ echo -n user:ff40aec-44c2-495a-bbbf-3e0751568de3 | base64
dXNlcjpmZjQwYWVlYy00NGMyLTQ5NWEtYmJiZi0zZTA3NTE1NjhkZTM=

$ curl -v -X GET http://localhost:8080/api/anonymous/hello?name=jim \
-H "Authorization: BASIC dXNlcjpmZjQwYWVlYy00NGMyLTQ5NWEtYmJiZi0zZTA3NTE1NjhkZTM="
> GET /api/anonymous/hello?name=jim HTTP/1.1
> Authorization: BASIC dXNlcjpmZjQwYWVlYy00NGMyLTQ5NWEtYmJiZi0zZTA3NTE1NjhkZTM=
>
< HTTP/1.1 200 ①
< Content-Length: 10
hello, jim
```

① request with successful BASIC authentication gives us the results of intended URL

Base64 web sites available if command-line tool not available



I am using a command-line tool for easy demonstration and privacy. There are various [websites](#) that will perform the encode/decode for you as well. Obviously, using a public website for real usernames and passwords would be a bad idea.



curl can Automatically Supply Authorization Header

You can avoid the manual step of base64 encoding the username:password and manually supplying the Authorization header with curl by using the plaintext -u username:password option.

```
curl -v -X GET http://localhost:8080/api/anonymous/hello?name=jim -u  
user:ff40aeec-44c2-495a-bbbf-3e0751568de3  
...  
> GET /api/anonymous/hello?name=jim HTTP/1.1  
> Authorization: BASIC  
dXNlcjpmZjQwYWVlYy00NGMyLTQ5NWEtYmJiZi0zZTA3NTE1NjhkZTM=  
...
```

194.3. Authentication Required Activated

If we do not supply the `Authorization` header or do not supply a valid value, we get a 401/UNAUTHORIZED status response back from the interface telling us our credentials are either invalid (did not match username:password) or were not provided.

Example Unauthorized Access

```
$ echo -n user:badpassword | base64 ②  
dXNlcjpiYWRwYXNzd29yZA==  
  
$ curl -v -X GET http://localhost:8080/api/anonymous/hello?name=jim -u  
user:badpassword ①  
> GET /api/anonymous/hello?name=jim HTTP/1.1  
> Authorization: BASIC dXNlcjpiYWRwYXNzd29yZA== ②  
>  
< HTTP/1.1 401  
< WWW-Authenticate: Basic realm="Realm"  
< Set-Cookie: JSESSIONID=32B6CDB8E899A82A1B7D55BC88CA5CBE; Path=/; HttpOnly  
< WWW-Authenticate: Basic realm="Realm"  
< Content-Length: 0
```

① bad `username:password` supplied

② demonstrating source of Authorization header

194.4. Username/Password can be Supplied

To make things more consistent during this stage of our learning, we can manually assign a username and password using properties.

`src/main/resources/application.properties`

```
spring.security.user.name: user  
spring.security.user.password: password
```

Example Authentication with Supplied Username/Password

```
$ curl -v -X GET "http://localhost:8080/api/authn/hello?name=jim" -u user:password
> GET /api/authn/hello?name=jim HTTP/1.1
> Authorization: BASIC dXNlcjpwYXNzd29yZA==
< HTTP/1.1 200
< Set-Cookie: JSESSIONID=7C5045AE82C58F0E6E7E76961E0AFF57; Path=/; HttpOnly
< Content-Length: 10
hello, jim
```

194.5. CSRF Protection Activated

The default Security Filter chain contains [CSRF protections](#)—which is a defense mechanism developed to prevent alternate site from providing the client browser a page that performs an unsafe (POST, PUT, or DELETE) call to an alternate site the client has an established session with. The server makes a CSRF token available to us using a GET and will be expecting that value on the next POST, PUT, or DELETE.

Example CSRF POST Rejection

```
$ curl -v -X POST "http://localhost:8080/api/authn/hello" \
-u user:password -H "Content-Type: text/plain" -d "jim"
> POST /api/authn/hello HTTP/1.1
> Authorization: BASIC dXNlcjpwYXNzd29yZA==
> Content-Type: text/plain
> Content-Length: 3
< HTTP/1.1 401 ①
< Set-Cookie: JSESSIONID=3EEB3625749482AD9E44A3B7E25A0EE4; Path=/; HttpOnly
< WWW-Authenticate: Basic realm="Realm"
< Content-Length: 0
```

① POST rejected because no CSRF token provided by client

194.6. Other Headers

Spring has, by default, generated additional headers to help with client interactions that primarily have to do with common security issues.

Example Other Headers Supplied By Spring

```
$ curl -v http://localhost:8080/api/anonymous/hello?name=jim -u user:password
> GET /api/anonymous/hello?name=jim HTTP/1.1
> Authorization: BASIC dXNlcjpwYXNzd29yZA==
>
< HTTP/1.1 200
< Vary: Origin
< Vary: Access-Control-Request-Method
< Vary: Access-Control-Request-Headers
< X-Content-Type-Options: nosniff
```

```
< X-XSS-Protection: 0
< Cache-Control: no-cache, no-store, max-age=0, must-revalidate
< Pragma: no-cache
< Expires: 0
< X-Frame-Options: DENY
< Content-Type: text/html; charset=UTF-8
< Content-Length: 10
< Date: Fri, 30 Aug 2024 22:16:48 GMT
<
hello, jim
```

Vary

indicates that content will be influenced by these headers. Intermediate/browser caches should especially use the Origin value when making a cache key. This prevents a different Origin from receiving a cached copy to the same resource and triggering a CORS error.

X-Content-Type-Options

informs the browser that supplied Content-Type header responses have been deliberately assigned ^[1] and to avoid Mime Sniffing (trying to determine for itself)—a problem caused by servers serving uploaded content meant to masquerade as alternate MIME types.

X-XSS-Protection

a header that informs the browser what to do in the event of a Cross-Site Scripting attack is detected. There seems to be a lot of skepticism of its value for certain browsers ^[2]

Cache-Control

a header that informs the client how the data may be cached. ^[3] This value can be set by the controller response but is set to a non-cache state by default here.

Pragma

an HTTP/1.0 header that has been replaced by Cache-Control in HTTP 1.1. ^[4]

Expires

a header that contains the date/time when the data should be considered stale and should be re-validated with the server. ^[5]

X-Frame-Options

informs the browser whether the contents of the page can be displayed in a frame. ^[5] This helps prevent site content from being hijacked in an unauthorized manner. This will not be pertinent to our API responses.

[1] "[X-Content-Type-Options](#)", MDN web docs

[2] "[X-XSS-Protection Header](#)", OWASP Cheat Sheet Series

[3] "[Cache-Control](#)", MDN web docs

[4] "[Pragma](#)", MDN web docs

[5] "[X-Frame-Options](#)", MDN web docs

Chapter 195. Default FilterChainProxy Bean

The above behavior was put in place by the default Security AutoConfiguration—which is primarily placed within an instance of the `FilterChainProxy` class ^[1]. This makes the `FilterChainProxy` class a convenient place for a breakpoint when debugging security flows.

The `FilterChainProxy` is configured with a set of firewall rules that address such things as bad URI expressions that have been known to hurt web applications and zero or more `SecurityFilterChains` arranged in priority order (first match wins).

The default configuration has a single `SecurityFilterChain` that matches all URIs, requires authentication, and also adds the other aspects we have seen so far.

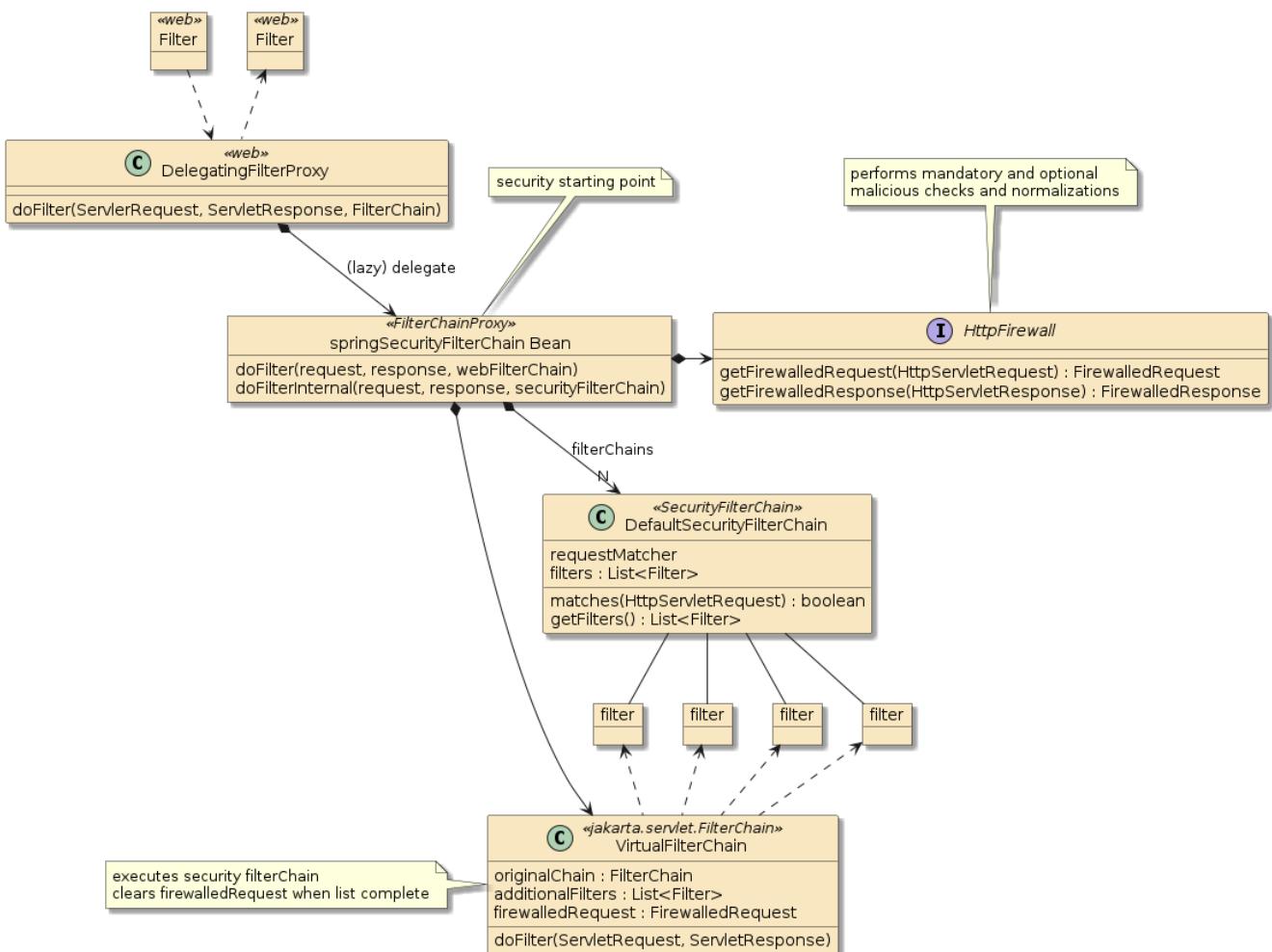


Figure 83. Spring FilterChainProxy Configuration

Below is a list of filters put in place by the default configuration. This—by far—is not all the available filters. I wanted to at least provide a description of the default ones before we start looking to selectively configure the chain.

It is a pretty dry topic to just list them off. It would be best if you had the `svc/svc-security/noauthn-security-example` example loaded in an IDE with:

- the pom updated to include the `spring-boot-starter-security`

Starter Activates Default Security Policies

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

- Logger for `org.springframework.security.web` set to `TRACE`
- a breakpoint set on "`FilterChainProxy.doFilterInternal()`" to clearly display the list of filters that will be used for the request.

```
209     private void doFilterInternal(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException, ServletException {
210         FirewalledRequest firewallRequest = this.firewall.getFirewalledRequest((HttpServletRequest) request);
211         HttpServletResponse firewallResponse = this.firewall.getFirewalledResponse((HttpServletResponse) response);
212         List<Filter> filters = getFilters(firewallRequest);
213         if (filters == null || filters.size() == 0) {
214             if (logger.isTraceEnabled()) {
215                 logger.trace(LogMessage.of(() -> "No security for " + requestLine(firewallRequest)));
216             }
217             firewallRequest.reset();
218             this.filterChainDecorator.decorate(chain).doFilter(firewallRequest, firewallResponse);
219             return;
220         }
221         if (logger.isDebugEnabled()) {
222             logger.debug("Scanning " + requestLine(firewallRequest));
223         }
224     }
```

Java Watch View:

- Evaluate expression (⌚) or add a watch (⌚)
- > `request.getRequestURI() = "/api/authn/hello"`
- > `((RequestFacade) request).getMethod() = "GET"`
- > `this = {FilterChainProxy@7011} "FilterChainProxy[Filter Chains: [DefaultSecurityFilterChain [RequestMatcher=any request, Filters=[org.springframework.security.web.s...`
- > `request = {RequestFacade@7005}`
- > `response = {ResponseFacade@7006}`
- > `chain = {CompositeFilter$VirtualFilterChain@7007}`
- > `firewallRequest = {StrictHttpFirewall$StrictFirewalledRequest@7008} "FirewalledRequest[org.apache.catalina.connector.RequestFacade@4438314d]"`
- > `firewallResponse = {FirewalledResponse@7009}`
- filters = {ArrayList@7010} size = 16**
 - > `0 = {DisableEncodeUrlFilter@8760}`
 - > `1 = {WebAsyncManagerIntegrationFilter@8761}`
 - > `2 = {SecurityContextHolderFilter@8762}`
 - > `3 = {HeaderWriterFilter@8763}`
 - > `4 = {CorsFilter@8764}`
 - > `5 = {CsrfFilter@8765}`
 - > `6 = {LogoutFilter@8766}`
 - > `7 = {UsernamePasswordAuthenticationFilter@8767}`
 - > `8 = {DefaultLoginPageGeneratingFilter@8768}`
 - > `9 = {DefaultLogoutPageGeneratingFilter@8769}`
 - > `10 = {BasicAuthenticationFilter@8770}`
 - > `11 = {RequestCacheAwareFilter@8771}`
 - > `12 = {SecurityContextHolderAwareRequestFilter@8772}`
 - > `13 = {AnonymousAuthenticationFilter@8773}`
 - > `14 = {ExceptionTranslationFilter@8774}`
 - > `15 = {AuthorizationFilter@8775}`

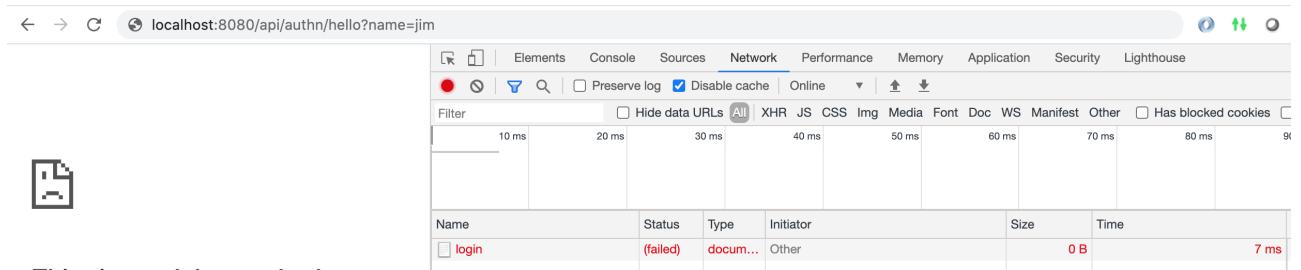
- another breakpoint set on "`FilterChainProxy.VirtualFilterChain.doFilter()`" to pause in between each filter.

```

361
362
363     @Override
364     public void doFilter(ServletRequest request, ServletResponse response) throws IOException, ServletException {
365         if (this.currentPosition == this.size) {
366             this.originalChain.doFilter(request, response); originalChain: FilterChainProxy$lambda@8387
367             return;
368         }
369         this.currentPosition++;
370         Filter nextFilter = this.additionalFilters.get(this.currentPosition - 1); nextFilter: DisableEncodeUrlFilter
371         if (logger.isTraceEnabled()) {
372             String name = nextFilter.getClass().getSimpleName();
373             logger.trace(LogMessage.format("Invoking %s (%d/%d)", name, this.currentPosition, this.size));
374         }
375     }

```

- a browser open with network logging active and ready to navigate to <http://localhost:8080/api/authn/hello?name=jim>



Whenever we make a request in the default state - we will most likely visit the following filters.

DisableEncodedUrlFilter

An exit filter that cleanses URLs provided in the response header. It specifically looks for and removes session IDs, which are not technically part of the URL and can be leaked into HTTP access logs.

WebAsyncManagerIntegrationFilter

Establishes an association between the SecurityContext (where the current caller's credentials are held) and potential async responses making use of the `Callable` feature. Caller identity is normally unique to a thread and obtained through a `ThreadLocal`. Anything completing in an alternate thread must have a strategy to resolve the identity of this user by some other means.

SecurityContextHolderFilter

Manages SecurityContext in between calls. If appropriate—stores the SecurityContext and clears it from the call on exit. If present—restores the SecurityContext on following calls.

HeaderWriterFilter

Issues standard headers (shown earlier) that can normally be set to a fixed value and optionally overridden by controller responses.

CorsFilter

Implements Cross-Origin Resource Sharing restrictions that allow or disable exchange of information with Web clients originating from a different DNS origin (schema, domain, port). This is the source of `Vary` and `Access-Control-Allow-Origin` headers.

CsrfFilter

Checks all non-safe (POST, PUT, and DELETE) calls for a special Cross-Site Request Forgery (CSRF) token either in the payload or header that matches what is expected for the session. This attempts to make sure that anything that is modified on this site — came from this site and not a malicious source. This does nothing for all safe (GET, HEAD, OPTIONS, and TRACE) methods.

LogoutFilter

Looks for calls to log out URI. If matches, it ends the login for all types of sessions, and terminates the chain.

UsernamePasswordAuthenticationFilter

This instance of this filter is put in place to obtain the username and password submitted by the login page. Therefore, anything that is not `POST /login` is ignored. The actual `POST /login` requests have their username and password extracted, authenticated

DefaultLoginPageGeneratingFilter

Handles requests for the login URI (`POST /login`). This produces the login page, terminates the chain, and returns to caller.

DefaultLogoutPageGeneratingFilter

Handles requests for the logout URI (`GET /logout`). This produces the logout page, terminates the chain, and returns to the caller.

BasicAuthenticationFilter

Looks for BASIC Authentication header credentials, performs authentication, and continues the flow if successful or if no credentials where present. If credentials were not successful it calls an authentication entry point that handles a proper response for BASIC Authentication and ends the flow.



Authentication Issues?

If you are having issues with authentication, a breakpoint on line 231 of `AbstractAuthenticationProcessingFilter` would be a good place to start.

RequestCacheAwareFilter

This retrieves an original request that was redirected to a login page and continues it on that path.

SecurityContextHolderAwareRequestFilter

Wraps the `HttpServletRequest` so that the security-related calls (`isAuthenticated()`, `authenticate()`, `login()`, `logout()`) are resolved using the Spring security context.

AnonymousAuthenticationFilter

Populates the Spring security context with anonymous principal if no user is identified

ExceptionTranslationFilter

Attempts to map any thrown `AccessDeniedException` and `AuthenticationException` to an HTTP Response. It does not add any extra value if those exceptions are not thrown. This will save the

current request (for access by RequestCacheAwareFilter) and commence an authentication for AccessDeniedExceptions if the current user is anonymous. The saved current request will allow the subsequent login to complete with a resumption of the original target. If FORM Authentication is active—the commencement will result in a 302/REDIRECT to the `/login` URI.

AuthorizationFilter

Restricts access based upon the response from the assigned `AuthorizationManager`.



Authorization Issues?

If you are having issues with authentication, line 95 of `AuthorizationFilter` is also a useful place to create a breakpoint.

Successfully reaching the end of the Security Filter Chain, this is where the SecurityFilter chain hands control back to the original ApplicationFilterChain, where the endpoint will get invoked.

Security FilterChain Handing Control Back to Application FilterChain

```
public void doFilter(ServletRequest request, ServletResponse response) throws  
IOException, ServletException {  
    if (this.currentPosition == this.size) { ①  
        this.originalChain.doFilter(request, response); ②  
        return;  
    }  
}
```

① successfully reached the end of the SecurityFilter chain

② delegating control back to the Application Filter chain

[1] "FilterChainProxy", Spring Security Reference Manual

Chapter 196. Summary

In this module, we learned:

1. the importance of identity, authentication, and authorization within security
2. the purpose for and differences between encoding, encryption, and cryptographic hashes
3. purpose of a filter-based processing architecture
4. the identity of the core components within Spring Authentication
5. where the current user authentication is held/located
6. how to activate default Spring Security configuration
7. the security features of the default Spring Security configuration
8. to step through a series of calls through the Security filter chain for the ability to debug future access problems

Spring Security Authentication

copyright © 2024 jim stafford (jim.stafford@jhu.edu)

Chapter 197. Introduction

In the previous example we accepted all defaults and inspected the filter chain and API responses to gain an understanding of the Spring Security framework. In this chapter we will begin customizing the authentication configuration to begin to show how and why this can be accomplished.

197.1. Goals

You will learn:

- to create a customized security authentication configurations
- to obtain the identity of the current, authenticated user for a request
- to incorporate authentication into integration tests

197.2. Objectives

At the conclusion of this lecture and related exercises, you will be able to:

1. create multiple, custom authentication filter chains
2. enable open access to static resources
3. enable anonymous access to certain URIs
4. enforce authenticated access to certain URIs
5. locate the current authenticated user identity
6. enable Cross-Origin Resource Sharing (CORS) exchanges with browsers
7. add an authenticated identity to RestTemplate and RestClient clients
8. add authentication to integration tests

Chapter 198. Configuring Security

To override security defaults and define a customized `FilterChainProxy`-- we must supply one or more classes that define our own `SecurityFilterChain(s)`.

198.1. WebSecurityConfigurer and Component-based Approaches

Spring has provided two ways to do this:

- `WebSecurityConfigurer`/ `WebSecurityConfigurerAdapter` - is the legacy, deprecated (Spring Security 5.7.0-M2; 2022), and later removed (Spring 6) definition class that acts as a modular factory for security aspects of the application.^[1] There can be one-to-N `WebSecurityConfigurers` and each can define a `SecurityFilterChain` and supporting services.
- `Component-based configuration` - is the modern approach to defining security aspects of the application. The same types of components are defined with the component-based approach, but they are instantiated in independent `@Bean` factories. Any interdependency between the components is wired up using beans injected into the `@Bean` factory.

Early versions of Spring were based solely on the `WebSecurityConfigurer` method. Later versions of Spring 5 provided support for both. Spring 6 now only supports the Component-based method. Since you will likely encounter the `WebSecurityConfigurer` approach for a long while in older applications, I will provide some coverage of that here. However, the main focus will be the Spring 6 Component-based approach. Refer to older versions of the course examples and notes for more focused coverage of the `WebSecurityConfigurer` approach.

To highlight that the `FilterChainProxy` is populated with a prioritized list of `SecurityFilterChain`—I am going to purposely create multiple chains.

- one with the API rules (`APIConfiguration`) - highest priority
- one with the former default rules (`AltConfiguration`) - lowest priority
- one with access rules for Swagger (`SwaggerSecurity`) - medium priority

The priority indicates the order in which they will be processed and will also influence the order for the `SecurityFilterChain`s they produce. Normally I would not highlight Swagger in these examples—but it provides an additional example of how we can customize Spring Security.

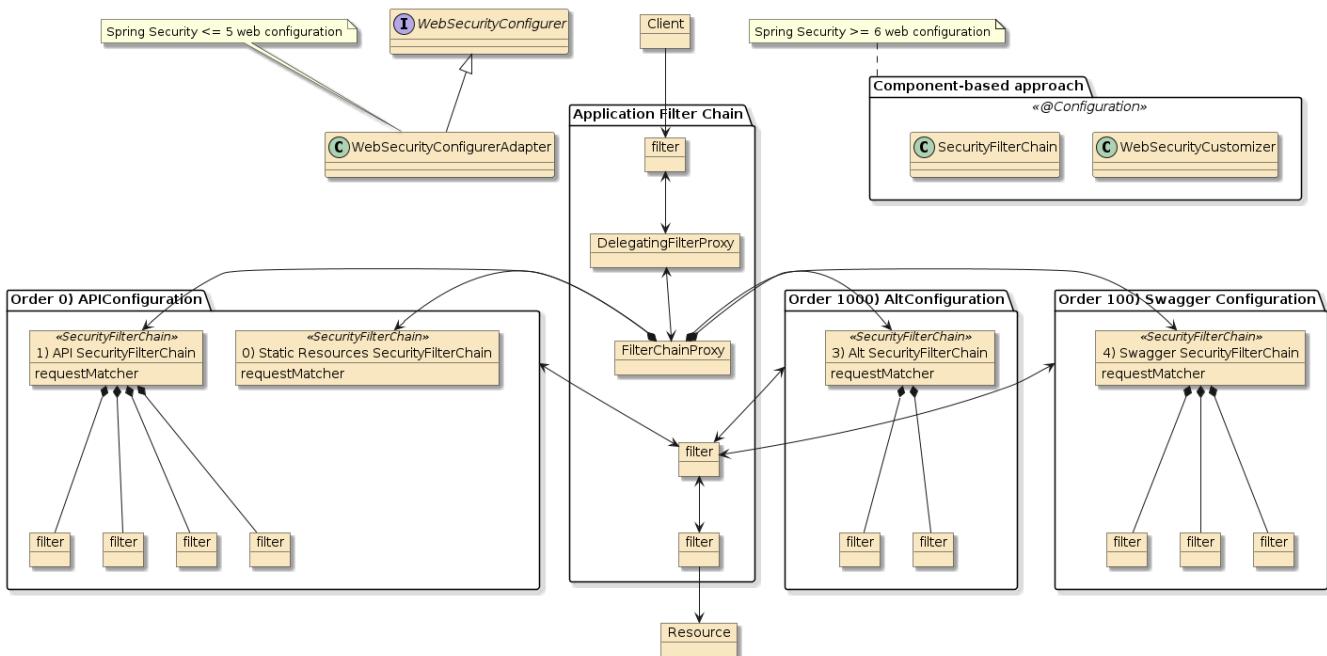


Figure 84. Multiple SecurityFilterChains

198.2. Core Application Security Configuration

The example will eventually contain several `SecurityFilterChains`, but let's start with focusing on just one of them—the "API Configuration". This initial configuration will define the configuration for access to static resources, dynamic resources, and how to authenticate our users.

198.2.1. WebSecurityConfigurerAdapter Approach

In the legacy `WebSecurityConfigurer` approach, we would start by defining a `@Configuration` class that extends `WebSecurityConfigurerAdapter` and overrides one or more of its configuration methods.



Legacy WebSecurityConfigurer no Longer Exists

Reminder - the legacy `WebSecurityConfigurer` approach does not exist in Spring Boot 3 / Spring Security 6. It is included here as an aid for those that may be transitioning from a legacy baseline.

WebSecurityConfigurer Approach

```

@Configuration(proxyBeanMethods = false)
@Order(0) ②
public class APIConfiguration extends WebSecurityConfigurerAdapter { ①
    @Override
    public void configure(WebSecurity web) throws Exception { ... } ③
    @Override
    protected void configure(HttpSecurity http) throws Exception { ... } ④
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception { ... }
} ⑤
@Bean
@Override

```

```
public AuthenticationManager authenticationManager() throws Exception { ... }
```

⑥

- ① Create `@Configuration` class that extends `WebSecurityConfigurerAdapter` to customize `SecurityFilterChain`
- ② `APIConfiguration` has a high priority resulting `SecurityFilterChain` for dynamic resources
- ③ configure a `SecurityFilterChain` for static web resources
- ④ configure a `SecurityFilterChain` for dynamic web resources
- ⑤ optionally configure an `AuthenticationManager` for multiple authentication sources
- ⑥ optionally expose `AuthenticationManager` as an injectable bean for re-use in other `SecurityFilterChains`

Each `SecurityFilterChain` would have a reference to its `AuthenticationManager`. The `WebSecurityConfigurerAdapter` provided the chance to custom configure the `AuthenticationManager` using a builder.

The adapter also provided an accessor method that exposed the built `AuthenticationManager` as a pre-built component for other `SecurityFilterChains` to reference.

198.2.2. Component-based Approach

In the modern Component-based approach, we define each aspect of our security infrastructure as a separate component. These `@Bean` factory methods are within a normal `@Configuration` class that requires no inheritance. The names of the `@Bean` factory methods have no significance as long as they are unique. Only what they return has significance.

Component-based Approach

```
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.config.annotation.web.configuration.WebSecurityCustomizer
;
import org.springframework.security.web.SecurityFilterChain;
...

@Bean
public WebSecurityCustomizer apiStaticResources() { ... } ①
@Bean
@Order(Ordered.HIGHEST_PRECEDENCE) // ③
public SecurityFilterChain apiSecurityFilterChain(HttpSecurity http) throws Exception
{ ... } ②
@Bean
public AuthenticationManager authnManager(HttpSecurity http, ...) throws Exception {
⑤
    AuthenticationManagerBuilder builder = http ④
        .getSharedObject(AuthenticationManagerBuilder.class);
    ...
}
```

- ① define a bean to configure a `SecurityFilterChain` for static web resources
- ② define a bean to configure a `SecurityFilterChain` for dynamic web resources
- ③ high priority assigned to `SecurityFilterChain`
- ④ optionally configure an `AuthenticationManager` for multiple authentication sources
- ⑤ expose `AuthenticationManager` as an injectable bean for use in `SecurityFilterChains`

The `SecurityFilterChain` for static resources gets defined within a lambda function implementing the `WebSecurityCustomizer` interface. The `SecurityFilterChain` for dynamic resources gets directly defined by within the `@Bean` factory method.

There is no longer any direct linkage between the configuration of the `AuthenticationManager` and the `SecurityFilterChains` being built. The linkage is provided through a `getSharedObject` call of the `HttpSecurity` object that can be injected into the bean methods.

198.3. Ignoring Static Resources

One of the easiest rules to put into place is to provide open access to static content. This is normally image files, web CSS files, etc. Spring recommends **not** including dynamic content in this list. Keep it limited to static files.

Access is defined by configuring the `WebSecurity` object.

- In the legacy `WebSecurityConfigurerAdapter` approach, the modification was performed within the method overriding the `configure(WebSecurity)` method. Note that Spring 5 `WebSecurity` interface contained builder methods for `RequestMatcher` (e.g., `antMatchers()`) that no longer exist.

Ignore Static Content Configuration - Legacy WebSecurityConfigurerAdapter approach

```
import org.springframework.security.config.annotation.web.builders.WebSecurity;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

@Configuration
@Order(0)
public class APIConfiguration extends WebSecurityConfigurerAdapter {
    @Override
    public void configure(WebSecurity web) throws Exception {
        web.ignoring().antMatchers("/content/**");
    }
}
```

- In the modern Component-based approach, a lambda function implementing the `WebSecurityCustomizer` functional interface is returned. That lambda will be called to customize the `WebSecurity` object. Spring 6 made a breaking change to the `WebSecurity` interface by removing the approach-specific builders for matchers (e.g., `antMatchers()`) and generalized the call to `requestMatchers()`. Under the hood, by default, Spring will create a matcher that best suites the runtime environment—which will be `MvcRequestMatcher` in Spring MVC environments.

```
import  
org.springframework.security.config.annotation.web.configuration.WebSecurityCustomi  
zer;  
  
@Bean  
public WebSecurityCustomizer apiStaticResources() {  
    return (web)->web.ignoring().requestMatchers("/content/**"); ①  
}
```

① delegating to Spring to create the correct matcher

WebSecurityCustomers Functional Interface

```
public interface WebSecurityCustomizer {  
    void customize(WebSecurity web);  
}
```



MvcRequestMatcher is used by Spring MVC to implement URI matching based on the actual URIs hosted within Spring MVC. The expression is still an Ant expression. It is just performed using more knowledge of the hosted resources within Spring MVC. The **MvcRequestMatcher** implementation is said to be less prone to definition error when using Spring MVC. Spring will revert to **AntRequestMatcher** when running in alternate Servlet environments (e.g., JAX-RS).



The generic **requestMatchers()** approach did not work when the application contained a blend of Spring MVC and non-Spring MVC (e.g., H2 Database UI) servlets. In those cases, we have to use an explicit approach documented in [CVE-2023-34035](#). This has since been fixed.

Remember—our static content is packaged within the application by placing it under the **src/main/resources/static** directory of the source tree.

Static Content

```
$ tree src/main/resources/  
src/main/resources/  
|-- application.properties  
`-- static  
    '-- content  
        |-- hello.js  
        |-- hello_static.txt  
        '-- index.html  
$ cat src/main/resources/static/content/hello_static.txt  
Hello, static file
```

With that rule in place, we can now access our static file without any credentials.

```
$ curl -v -X GET http://localhost:8080/content/hello_static.txt
> GET /content/hello_static.txt HTTP/1.1
>
< HTTP/1.1 200
< Vary: Origin
< Vary: Access-Control-Request-Method
< Vary: Access-Control-Request-Headers
< Last-Modified: Fri, 03 Jul 2020 19:36:25 GMT
< Cache-Control: no-store
< Accept-Ranges: bytes
< Content-Type: text/plain
< Content-Length: 19
< Date: Fri, 03 Jul 2020 20:55:58 GMT
<
Hello, static file
```

198.4. SecurityFilterChain Matcher

The meat of the `SecurityFilterChain` definition is within the configuration of the `HttpSecurity` object.

The resulting `SecurityFilterChain` will have a `RequestMatcher` that identifies which URIs the identified rules apply to. The default is to match "any" URI. In the example below I am limiting the configuration to URIs at and below `/api/anonymous` and `/api/authn`. The matchers also allow a specific HTTP method to be declared in the definition.

- In the legacy `WebSecurityConfigurerAdapter` approach, configuration is performed in the method overriding the `configure(HttpSecurity)` method. The legacy `HttpSecurity` interface provided builders that directly supported building different types of matchers (e.g., `antMatchers()`).

SecurityFilterChain Matcher - WebSecurityConfigurerAdapter approach

```
import org.springframework.security.config.annotation.web.builders.HttpSecurity;

@Configuration
@Order(0)
public class APIConfiguration extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.requestMatchers(m->m.antMatchers("/api/anonymous/**", "/api/authn/**"))
    );①
        //... ②
    }
    ...
}
```

① rules within this configuration will apply to URIs below `/api/anonymous` and `/api/authn`

② `http.build()` is **not** called



This method returns `void` and the `build()` method of `HttpSecurity` should **not** be called.

- In the modern Component-based approach, the configuration is performed in a `@Bean` method that will directly return the `SecurityFilterChain`. It has the same `HttpSecurity` object injected, but note that `build()` is called within this method to return a `SecurityFilterChain`. Spring 6 made a breaking changes to:

- the `HttpSecurity` interface by changing `requestMatchers()` to `securityMatchers()` in order to better distinguish between the pattern matching (`requestMatchers`) versus the role of the pattern matching (`securityMatchers`). The `securityMatchers()` method defines the aggregate `RequestMatchers` that are used to identify which calls invoke this `SecurityFilterChain`
- the `AbstractRequestMatcherRegistry` interface by removing type-specific matcher builders (e.g., removed `antMatchers()`) and generalized to `requestMatchers()`

Component-based HttpSecurity Configuration

```
@Bean  
 @Order(0)  
 public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {  
     http.securityMatchers(m->m.requestMatchers("/api/anonymous/**", "/api/authn/**")  
 );①  
     //...  
     return http.build(); ②  
 }
```

① rules within this configuration will apply to URIs below `/api/anonymous` and `/api/authn`

② `http.build()` is required for this `@Bean` factory



This method returns the `SecurityFilterChain` result of **calling** the `build()` method of `HttpSecurity`. This is different from the deprecated approach.

198.5. SecurityFilterChain with Explicit MvcRequestMatcher

The following lists the same solution using an explicit `MvcRequestMatcher`. The `MvcRequestMatcher` must be injected as a component because it requires knowledge of the servlet environment.

Component-based HttpSecurity Configuration with Explicit MvcRequestMatcher

```
import org.springframework.web.servlet.handler.HandlerMappingIntrospector;  
  
 @Bean  
 MvcRequestMatcher.Builder mvc(HandlerMappingIntrospector introspector) {  
     return new MvcRequestMatcher.Builder(introspector);  
 }
```

```

@Bean
@Order(Ordered.HIGHEST_PRECEDENCE) //0
public SecurityFilterChain apiSecurityFilterChain(HttpSecurity http,
    MvcRequestMatcher.Builder mvc) throws Exception {
    http.securityMatchers(m->m.requestMatchers(
        mvc.pattern("/api/anonymous/**"),
        mvc.pattern("/api/authn/**")
    ));
}

```

198.6. HttpSecurity Builder Methods

The `HttpSecurity` object is "builder-based" and has several options on how it can be called.

- `http.methodAcceptingBuilt(builtObject)`
- `http.methodReturningBuilder().customizeBuilder()` (non-lambda)
- `http.methodPassingBuilderToLambda(builder->builder.customizeBuilder())`

198.6.1. Deprecated non-Lambda Methods

Spring Security 6 has deprecated the non-lambda customize approach (for removal in Spring Security 7) in favor of the lambda approach.^[2] The following non-lambda approach (supplying no options) will result in a deprecation warning.

Deprecated non-Lambda Customizer Approach

```
http.httpBasic();
```

Spring Security has added a `Customizer.withDefaults()` construct that can mitigate the issue when supplying no customizations.

Lambda Customization Approach

```

import org.springframework.security.config.Customizer;

http.httpBasic(Customizer.withDefaults());

```

198.6.2. Chaining not Necessary

The builders are also designed to be chained. It is quite common to see the following syntax used.

Chained Builder Calls

```

http.authorizeRequests(cfg->cfg.anyRequest().authenticated())
    .formLogin(Customizer.withDefaults())
    .httpBasic(Customizer.withDefaults());

```

As much as I like chained builders—I am not a fan of that specific syntax when starting out. Especially if we are experimenting and commenting/uncommenting configuration statements. We can simply make separate calls. You will see me using separate calls. Either style functionally works the same.

Separate Builder Calls with Lambdas

```
http.authorizeRequests(cfg->cfg.anyRequest().authenticated());  
http.formLogin(Customizer.withDefaults());  
http.httpBasic(Customizer.withDefaults());
```

198.7. Match Requests

We first want to scope our `HttpSecurity` configuration commands using one of the `securityMatcher` methods. We can provide a `RequestMatcher` using `securityMatcher()` or a configururer using `securityMatchers()`. The following will create a `RequestMatcher` that will match any HTTP method and URI below `/api/anonymous` and `/api/authn`. We can add an `Http.(METHOD)` if we want it to be specific to only that HTTP method.

Using Default Request Matchers

```
http.securityMatchers(m->m.requestMatchers("/api/anonymous/**", "/api/authn/**")); ①  
//http.securityMatcher("/api/anonymous/**", "/api/authn/**"); ②
```

① matches all HTTP methods matching the given URI patterns

② shortcut when no HttpMethod is needed

RequestMatchers can be Aggregated



The code above technically creates three (3) `RequestMatchers`. Two (2) to match the individual (method)/URIs and one (1) parent to aggregate them into an "or" predicate.

The `requestMatchers()` configurer (*in non-ambiguous environments) will create an `MvcRequestMatcher` under the hood when Spring MVC is present and an `AntRequestMatcher` for other servlet frameworks. The `MvcRequestMatcher` is said to be less prone to error (i.e., missed matches) than other techniques. Spring 6 took away builder methods to directly create the `AntRequestMatcher` (and other explicit types), but they can still be created manually. This technique can be used to create [any type of RequestMatcher](#).

Using Explicit Request Matchers

```
import org.springframework.security.web.util.matcher.OrRequestMatcher;  
import org.springframework.security.web.util.matcher.RegexRequestMatcher;  
  
http.securityMatcher(new OrRequestMatcher(②  
    RegexRequestMatcher.regexMatcher("^/api/anonymous$"), ①  
    RegexRequestMatcher.regexMatcher("^/api/anonymous/.*"),  
    RegexRequestMatcher.regexMatcher("^/api/authn$"),
```

```

    RegexRequestMatcher.regexMatcher("^/api/authn/.*")
));

```

- ① supplying an explicit type of `RequestMatcher`
- ② Spring Security is aggregating the list into an `OrRequestMatcher`

Notice the `requestMatcher` is the primary property in the individual chains and the rest of the configuration is impacting the filters within that chain.

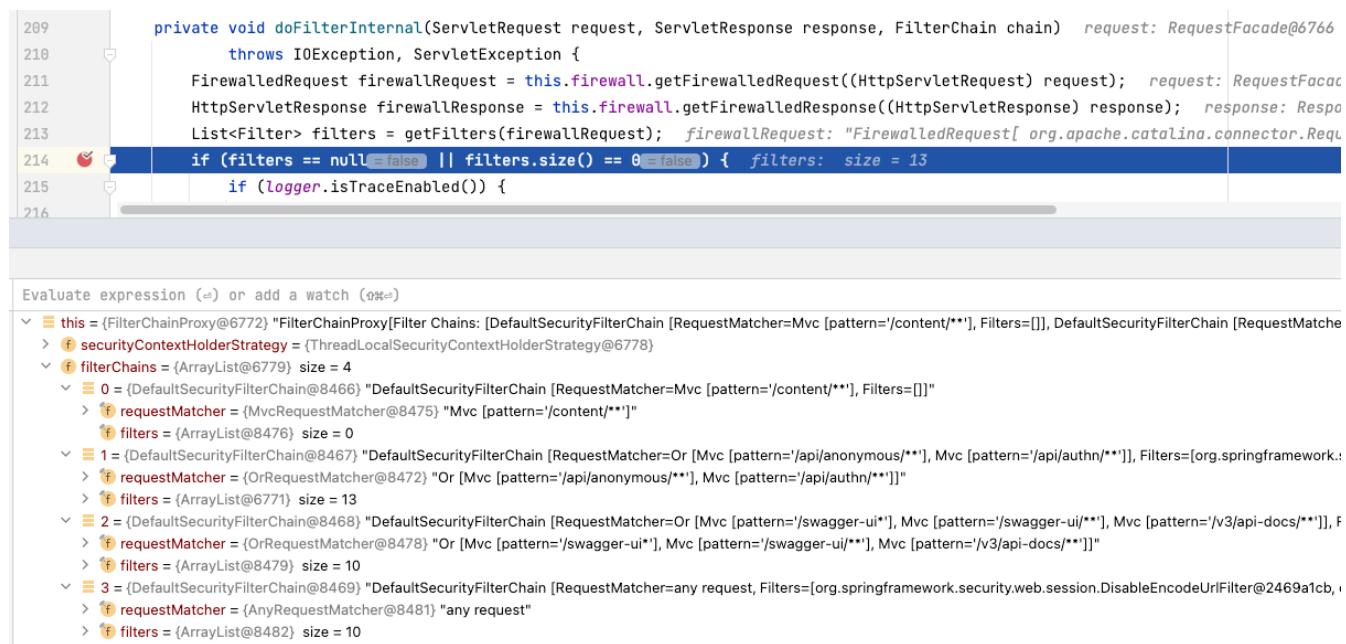


Figure 85. Request Matchers



Notice also that our initial `SecurityFilterChain` is within the other chains in the example and is high in priority because of our `@Order` value assignment:

198.8. Authorize Requests

Next I am showing the authentication requirements of the `SecurityFilterChain`. Calls to the `/api/anonymous` URIs do not require authentication. Calls to the `/api/authn` URIs do require authentication.

Defining Authentication Requirements

```

http.authorizeHttpRequests(cfg->cfg.requestMatchers("/api/anonymous/**").permitAll());
http.authorizeHttpRequests(cfg->cfg.anyRequest().authenticated());

```

The permissions off the matcher include:

- `permitAll()` - no constraints
- `denyAll()` - nothing will be allowed
- `authenticated()` - only authenticated callers may invoke these URIs
- role restrictions that we won't be covering just yet

You can also make your matcher criteria method-specific by adding in a `HttpMethod` specification.

Matchers Also Supports HttpMethod Criteria

```
import org.springframework.http.HttpMethod;  
...  
...(cfg->cfg.requestMatchers(HttpMethod.GET, "/api/anonymous/**").permitAll())  
...(cfg->cfg.requestMatchers(HttpMethod.GET).permitAll())
```

requestMatchers are evaluated after satisfying securityMatcher(s)



RequestMatchers are evaluated within the context of what satisfied the filter's securityMatcher(s). If you form a Security FilterChain for a specific base URI, the requestMatcher is only defining rules for what that chain processes.

198.9. Authentication

In this part of the example, I am enabling BASIC Auth and eliminating FORM-based authentication. For demonstration only—I am providing a custom name for the [realm name returned to browsers](#).

```
http.httpBasic(cfg->cfg.realmName("AuthConfigExample")); ①  
http.formLogin(cfg->cfg.disable());
```



Realm name is not a requirement to activate Basic Authentication. It is shown here solely as an example of something easily configured.

Example Realm Header used in Authentication Required Response

```
< HTTP/1.1 401  
< WWW-Authenticate: Basic realm="AuthConfigExample" ①
```

① Realm Name returned in HTTP responses requiring authentication

198.10. Header Configuration

In this portion of the example, I am turning off two of the headers that were part of the default set: XSS protection and frame options. There seemed to be some debate on the value of the XSS header^[1] and we have no concern about frame restrictions. By disabling them—I am providing an example of what can be changed.

CSRF protections have also been disabled to make non-safe methods more sane to execute at this time. Otherwise, we would be required to supply a value in a POST that came from a previous GET (all maintained and enforced by optional filters).

Header Configuration

```
http.headers(cfg->{
```

```
    cfg.xssProtection(xss->xss.disable());
    cfg.frameOptions(fo->fo.disable());
});
http.csrf(cfg->cfg.disable());
```

198.11. Stateless Session Configuration

I have no interest in using the Http Session to maintain identity between calls—so this should eliminate the **SET-COOKIE** commands for the **JSESSIONID**.

Stateless Session Configuration

```
http.sessionManagement(cfg->
    cfg.sessionCreationPolicy(SessionCreationPolicy.STATELESS));
```

[1] "Spring Security without the WebSecurityConfigurerAdapter", Spring.io, Feb 21, 2022

[2] "Spring Security, Preparing for 7.0, Configuration Methods", Spring.io

[3] "X-XSS-Protection", MDN Web Docs

Chapter 199. Configuration Results

With the above configurations in place—we can demonstrate the desired functionality and trace the calls through the filter chain if there is an issue.

199.1. Successful Anonymous Call

The following shows a successful anonymous call and the returned headers. Remember that we have gotten rid of several unwanted features with their headers. The controller method has been modified to return the identity of the authenticated caller. We will take a look at that later—but know the source of the additional `:caller=` string was added for this wave of examples.

Successful Anonymous Call

```
$ curl -v -X GET http://localhost:8080/api/anonymous/hello?name=jim
> GET /api/anonymous/hello?name=jim HTTP/1.1
< HTTP/1.1 200
< X-Content-Type-Options: nosniff
< Cache-Control: no-cache, no-store, max-age=0, must-revalidate
< Pragma: no-cache
< Expires: 0
< Content-Type: text/plain;charset=UTF-8
< Content-Length: 25
< Date: Fri, 03 Jul 2020 22:11:11 GMT
<
hello, jim :caller=(null) ①
```

① we have no authenticated user

199.2. Successful Authenticated Call

The following shows a successful authenticated call and the returned headers.

Successful Authenticated Call

```
$ curl -v -X GET http://localhost:8080/api/authn/hello?name=jim -u user:password ①
> GET /api/authn/hello?name=jim HTTP/1.1
> Authorization: BASIC dXNlcjpwYXNzd29yZA==
< HTTP/1.1 200
< X-Content-Type-Options: nosniff
< Cache-Control: no-cache, no-store, max-age=0, must-revalidate
< Pragma: no-cache
< Expires: 0
< Content-Type: text/plain;charset=UTF-8
< Content-Length: 23
< Date: Fri, 03 Jul 2020 22:12:34 GMT
<
hello, jim :caller=user ②
```

① example application configured with username/password of **user/password**

② we have an authenticated user

199.3. Rejected Unauthenticated Call Attempt

The following shows a rejection of an anonymous caller attempting to invoke a URI requiring an authenticated user.

Rejected Unauthenticated Call Attempt

```
$ curl -v -X GET http://localhost:8080/api/authn/hello?name=jim ①
> GET /api/authn/hello?name=jim HTTP/1.1
< HTTP/1.1 401
< WWW-Authenticate: Basic realm="AuthConfigExample"
< X-Content-Type-Options: nosniff
< Cache-Control: no-cache, no-store, max-age=0, must-revalidate
< Pragma: no-cache
< Expires: 0
< Content-Type: application/json
< Transfer-Encoding: chunked
< Date: Fri, 03 Jul 2020 22:14:20 GMT
<
{"timestamp":"2020-07-03T22:14:20.816+00:00","status":401,
"error":"Unauthorized","message":"Unauthorized","path":"/api/authn/hello"}
```

① attempt to make anonymous call to authentication-required URI

Chapter 200. Authenticated User

Authenticating the identity of the caller is a big win. We likely will want their identity at some point during the call.

200.1. Inject UserDetails into Call

One option is to inject the `UserDetails` containing the username (and authorities) for the caller. Methods that can be called without authentication will receive the `UserDetails` if the caller provides credentials but must protect itself against a null value if actually called anonymously.

Injecting Caller Identity into Controller

```
import org.springframework.security.core.annotation.AuthenticationPrincipal;
import org.springframework.security.core.userdetails.UserDetails;
...
public String getHello(@RequestParam(name = "name", defaultValue = "you") String name,
                      @AuthenticationPrincipal UserDetails user) {
    return "hello, " + name + " :caller=" + (user==null ? "(null)" : user.getUsername());
}
```

200.2. Obtain SecurityContext from Holder

The other option is to look up the `UserDetails` through the `SecurityContext` stored within the `SecurityContextHolder` class. This allows any caller in the call flow to obtain the identity of the caller at any time.

Obtaining Caller Identity from SecurityContextHolder

```
import org.springframework.security.core.context.SecurityContextHolder;

public String getHelloAlt(@RequestParam(name = "name", defaultValue = "you") String name) {
    Authentication authentication = SecurityContextHolder.getContext()
        .getAuthentication();
    Object principal = null!=authentication ? authentication.getPrincipal() : "(null)";
    String username = principal instanceof UserDetails ?
        ((UserDetails)principal).getUsername() : principal.toString();
    return "hello, " + name + " :caller=" + username;
}
```

Chapter 201. Swagger BASIC Auth Configuration

Once we enabled default security on our application—we lost the ability to fully utilize the Swagger page. We did not have to create a separate `SecurityFilterChain` for just the Swagger endpoints—but doing so provides some nice modularity and excuse to further demonstrate Spring Security configurability.



Check Defaults

Spring Boot used to always apply a default filter with `authenticated()`, denying access to any URI lacking a matching `securityMatcher`. Spring Boot 3.3.2 no longer applies a default filter when one is provided (as we did above)—leaving it unevaluated, resulting in open access and other defaults. Swagger UI can be accessed in this setting, but the BASIC authentication within the page is inoperable. Verify default behavior in addition to the URIs of importance to your application.

I have added a separate security configuration for the OpenAPI and Swagger endpoints.

201.1. Swagger Authentication Configuration

The following configuration allows the OpenAPI and Swagger endpoints to be accessed anonymously and handle authentication within OpenAPI/Swagger.

- Swagger SecurityFilterChain using the legacy `WebSecurityConfigurerAdapter` approach

```
@Configuration(proxyBeanMethods = false)
@Order(100) ①
public class SwaggerSecurity extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.requestMatchers(cfg->cfg
            .antMatchers("/swagger-ui*", "/swagger-ui/**", "/v3/api-docs/**"));
        http.authorizeRequests(cfg->cfg.anyRequest().permitAll());
        http.csrf().disable();
    }
}
```

① Priority (100) is after core application (0) and prior to default rules (1000)

- Swagger SecurityFilterChain using the modern Component-based approach

```
@Bean
@Order(100) ①
public SecurityFilterChain swaggerSecurityFilterChain(HttpSecurity http) throws
Exception {
    http.securityMatchers(cfg->cfg
```

```

        .requestMatchers("/swagger-ui*", "/swagger-ui/**", "/v3/api-docs/**"));
    http.authorizeHttpRequests(cfg->cfg.anyRequest().permitAll());
    http.csrf(cfg->cfg.disable());
    return http.build();
}

```

① Priority (100) is after core application (0) and prior to default rules (1000)

201.2. Swagger Security Scheme

In order for Swagger to supply a username:password using BASIC Auth, we need to define a **SecurityScheme** for Swagger to use. The following bean defines the core object the methods will be referencing.

Swagger BASIC Auth Security Scheme

```

package info.ejava.examples.svc.authn;

import io.swagger.v3.oas.models.Components;
import io.swagger.v3.oas.models.OpenAPI;
import io.swagger.v3.oas.models.security.SecurityScheme;
import org.springframework.context.annotation.Bean;
...

@Bean
public OpenAPI customOpenAPI() {
    return new OpenAPI()
        .components(new Components()
            .addSecuritySchemes("basicAuth",
                new SecurityScheme()
                    .type(SecurityScheme.Type.HTTP)
                    .scheme("basic")));
}

```

The **@Operation** annotations can now reference the **SecuritySchema** to inform the SwaggerUI that BASIC Auth can be used against that specific operation. Notice too, that we needed to make the injected **UserDetails** optional—or even better—hidden from OpenAPI/Swagger since it is not part of the HTTP request.

Swagger Operation BASIC Auth Definition

```

package info.ejava.examples.svc.authn.authcfg.controllers;

import io.swagger.v3.oas.annotations.Operation;
import io.swagger.v3.oas.annotations.Parameter;

@RestController
public class HelloController {
...

```

```

@Operation(description = "sample authenticated GET",
           security = @SecurityRequirement(name="basicAuth")) ①
@RequestMapping(path="/api/authn/hello",
               method= RequestMethod.GET)
public String getHello(
    @RequestParam(name="name",defaultValue="you",required=false) String name,
    @Parameter(hidden = true) ②
    @AuthenticationPrincipal UserDetails user) {
    return "hello, " + name + " :caller=" + user.getUsername();
}

```

① added `@SecurityRequirement` to operation to express within OpenAPI that this call accepts Basic Auth

② Identified parameter as not applicable to HTTP callers

With the `@SecurityRequirement` in place, the Swagger UI provides a means to supply username/password for subsequent calls.

The screenshot shows the Swagger UI interface for an OpenAPI definition. At the top, there's a header bar with browser controls and the URL `localhost:8080/swagger-ui/index.html?config...` . Below the header, the title **OpenAPI definition v0 OAS3** is displayed, along with a link to `/v3/api-docs`. A dropdown menu labeled **Servers** shows the URL `http://localhost:8080 - Generated server url`. To the right of the servers dropdown, there is a green button labeled **Authorize** with a lock icon, which is circled in red.

Below the header, the **hello-controller** section is expanded, showing sample calls with security constraints. The section title is **hello-controller** and it says **demonstrates sample calls with security constraints**. There are six API endpoint entries:

- GET /api/anonymous/hello** (blue button)
- POST /api/anonymous/hello** (green button)
- GET /api/authn/hello** (blue button) with a lock icon to its right
- POST /api/authn/hello** (green button) with a lock icon to its right
- GET /api/alt/hello** (blue button) with a lock icon to its right
- POST /api/alt/hello** (green button) with a lock icon to its right

Figure 86. Swagger with BASIC Auth Configured

When making a call—Swagger UI adds the Authorization header with the previously entered credentials.

The screenshot shows the Swagger UI interface for a GET request to '/api/authn/hello'. The top bar indicates the method (GET), path, and a lock icon. Below the path, it says 'sample authenticated GET'. A 'Parameters' section shows a required query parameter 'name' set to 'jim'. There are 'Execute' and 'Clear' buttons. The 'Responses' section is collapsed. The 'Curl' section contains a command with a red box highlighting the 'Authorization' header line: 'curl -X GET "http://localhost:8080/api/authn/hello?name=jim" -H "accept: */*" -H "Authorization: Basic dXNlcjpwYXNzd29yZA=="'.

Figure 87. Swagger BASIC Auth Call

Chapter 202. CORS

There is one more important security filter to add to our list before we end, and it is complex enough to deserve its own section - Cross Origin Resource Sharing (CORS). Using this standard, browsers will supply the URL of the source of Javascript used to call the server—when coming from a different host domain—and look for a response from the server that indicates the source is approved. Without support for CORS, javascript loaded by browsers will not be able to call the API unless it was loaded from the same base URL as the API. That even includes local development (i.e., javascript loaded from file system cannot invoke <http://localhost:8080>). In today's modern web environments—it is common to deploy services independent of Javascript-based UI applications or to have the UI applications calling multiple services with different base URLs.

202.1. Default CORS Support

The following example shows the result of the default CORS configuration for Spring Boot/Web MVC. The server is ignoring the `Origin` header supplied by the client and does not return any CORS-related authorization for the browser to use the response payload.

CORS Inactive, Origin Header Ignored

```
$ curl -v http://localhost:8080/api/anonymous/hello?name=jim
> GET /api/anonymous/hello?name=jim HTTP/1.1
> Host: localhost:8080
>
< HTTP/1.1 200
hello, jim :caller=(null)

$ curl -v http://localhost:8080/api/anonymous/hello?name=jim -H "Origin:
http://127.0.0.1:8080"
> GET /api/anonymous/hello?name=jim HTTP/1.1
> Host: localhost:8080
> Origin: http://127.0.0.1:8080 ①
>
< HTTP/1.1 200
hello, jim :caller=(null)
```

① `Origin` header normally supplied by browser when coming from a different domain—ignored by server

The lack of headers does not matter for curl, but the CORS response does get evaluated when executed within a browser.

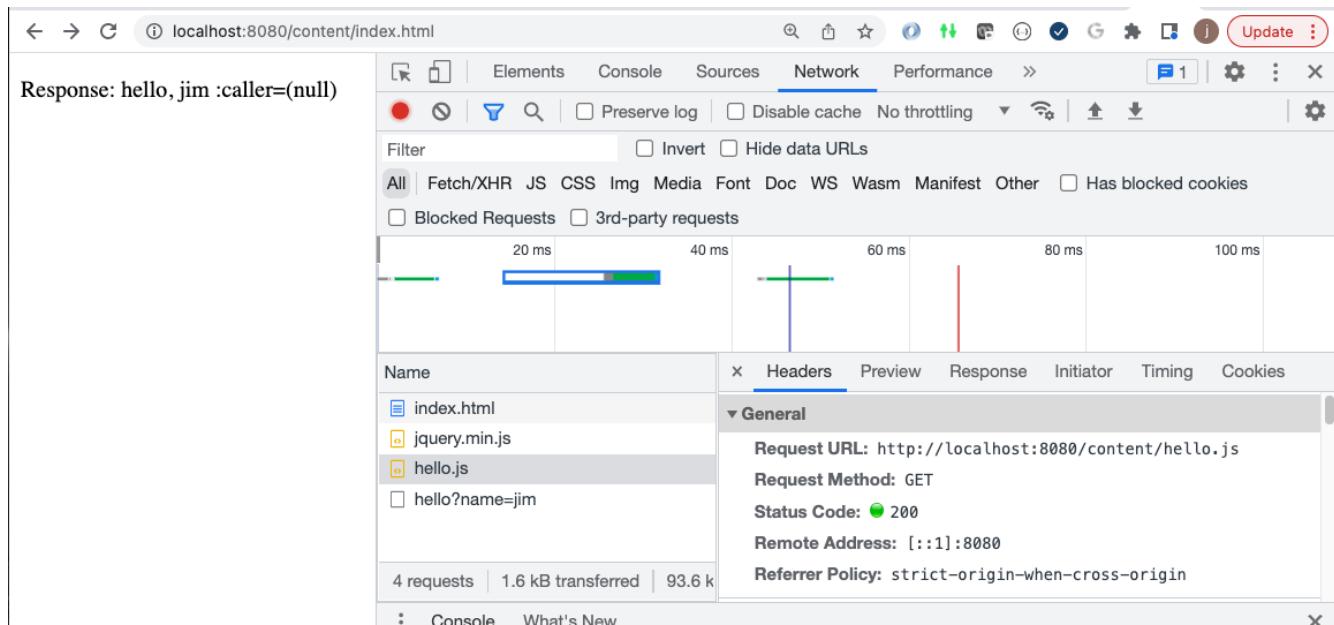
202.2. Browser and CORS Response

202.2.1. Same Origin/Target Host

The following is an example of Javascript loaded from <http://localhost:8080> and calling <http://localhost:8080>. No `Origin` header is passed by the browser because it knows the Javascript

was loaded from the same source it is calling.

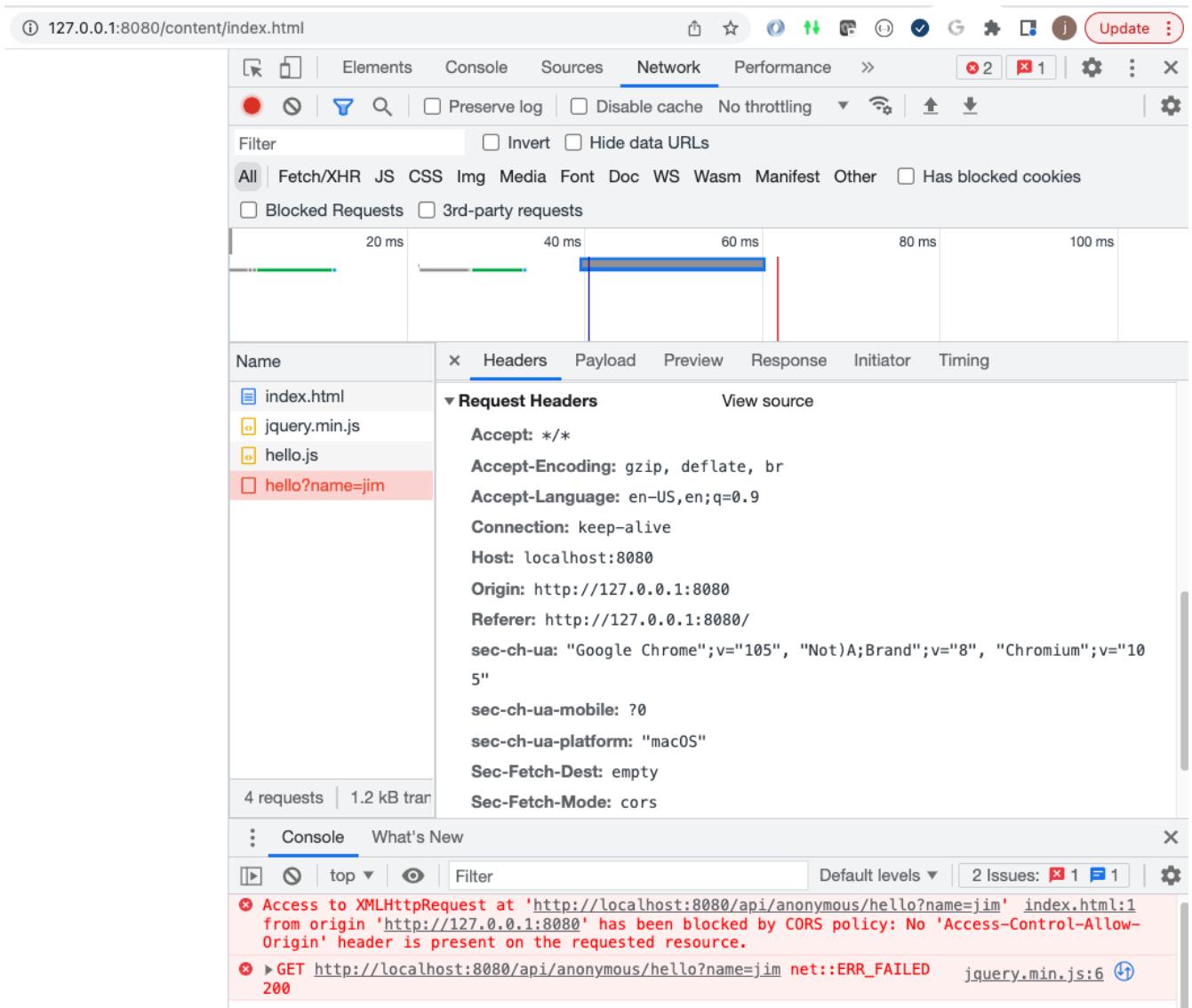
CORS Inactive, Origin Header Not Supplied for Same Source



202.2.2. Different Origin/Target Host

However, if we load the Javascript from an alternate source, the browser will fail to process the results. The following is an example of some Javascript loaded from `http://127.0.0.1:8080` and calling `http://localhost:8080`.

CORS Inactive, Origin Header Supplied for Different Source



202.3. Enabling CORS

To globally enable CORS support, we can invoke `http.cors(...)` with a method to call at runtime that will evaluate and return the result for the CORS request—based on a given `HttpServletRequest`. This is supplied when configuring the `SecurityFilterChain`.

Enabling CORS and Permit All

```
http.cors(cfg->cfg.configurationSource(corsPermitAllConfigurationSource()));
```

Example CORS Permit All Lambda Method Response

```
private CorsConfigurationSource corsPermitAllConfigurationSource() {
    return (request) -> {
        CorsConfiguration config = new CorsConfiguration();
        config.applyPermitDefaultValues();
        return config;
    };
}
```

CORS Evaluation Interface Implemented

```
package org.springframework.web.cors;

public interface CorsConfigurationSource {
    CorsConfiguration getCorsConfiguration(HttpServletRequest request);
}
```

202.3.1. CORS Headers

With CORS enabled and permitting all, we see some new VARY headers (indicating to caches that content will be influenced by these headers). The browser will be looking for the **Access-Control-Allow-Origin** header being returned with a value matching the **Origin** header passed in (* being a wildcard match).

CORS Approval Headers Returned in call cases

```
$ curl -v http://localhost:8080/api/anonymous/hello?name=jim
> GET /api/anonymous/hello?name=jim HTTP/1.1
> Host: localhost:8080 ①
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200
< Vary: Origin
< Vary: Access-Control-Request-Method
< Vary: Access-Control-Request-Headers ②
hello, jim :caller=(null)

$ curl -v http://localhost:8080/api/anonymous/hello?name=jim -H "Origin:
http://127.0.0.1:8080"
> GET /api/anonymous/hello?name=jim HTTP/1.1
> Host: localhost:8080
> Origin: http://127.0.0.1:8080 ③
>
< HTTP/1.1 200
< Vary: Origin
< Vary: Access-Control-Request-Method
< Vary: Access-Control-Request-Headers
< Access-Control-Allow-Origin: * ④
hello, jim :caller=(null)
```

① **Origin** header not supplied

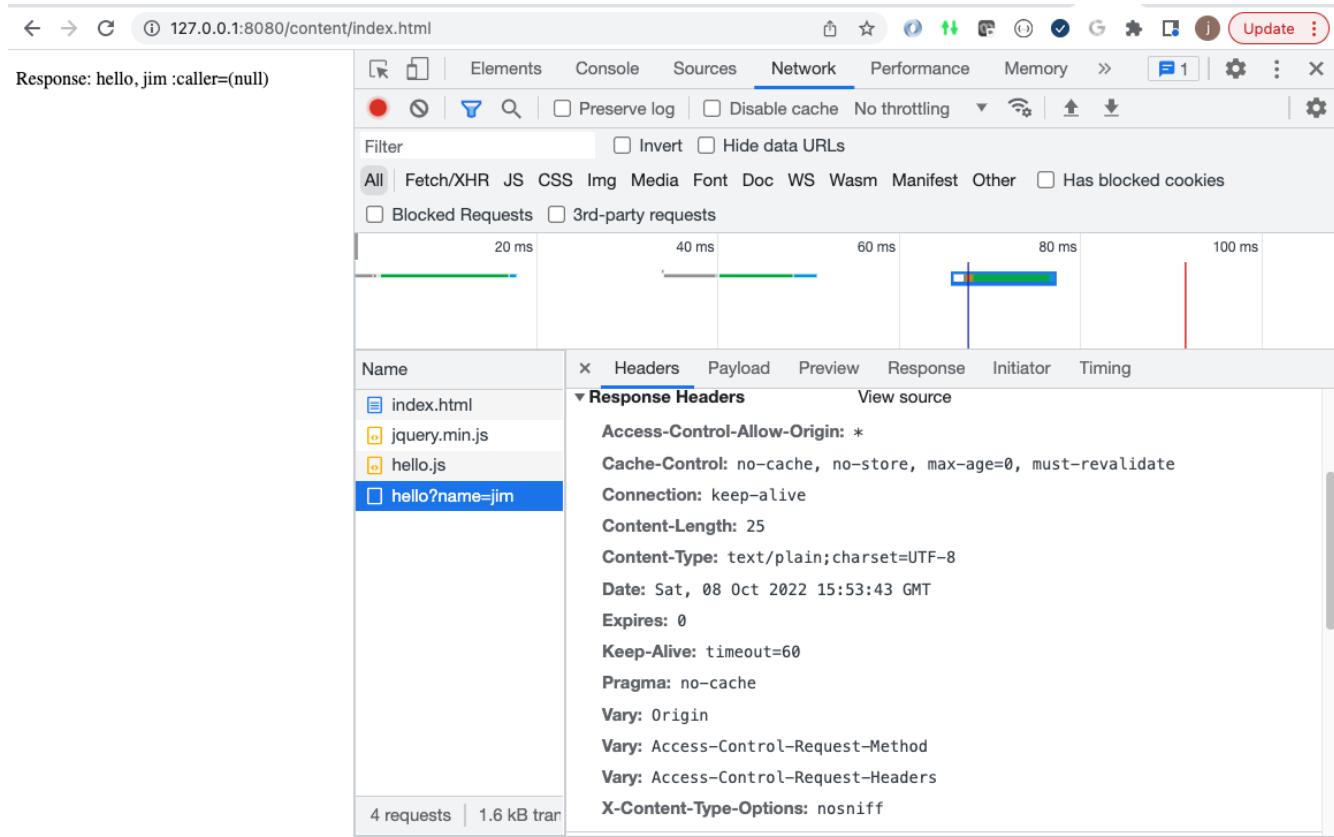
② No CORS **Access-Control-Allow-Origin** supplied in response

③ **Origin** header supplied by client

④ **Access-Control-Allow-Origin** denotes approval for the given Origin (* = wildcard)

202.3.2. Browser Accepts Access-Control-Allow-Origin Header

Browser Accepts CORS Access-Control-Allow-Origin Response



202.4. Constrained CORS

We can define more limited rules for CORS acceptance by using additional commands of the `CorsConfiguration` object.

Limiting CORS Acceptance

```
private CorsConfigurationSource corsLimitedConfigurationSource() {  
    return (request) -> {  
        CorsConfiguration config = new CorsConfiguration();  
        config.addAllowedOrigin("http://localhost:8080");  
        config.setAllowedMethods(List.of("GET", "POST"));  
        return config;  
    };  
}
```

202.5. CORS Server Acceptance

In this example, I have loaded the Javascript from `http://127.0.0.1:8080` and making a call to `http://localhost:8080` in order to match the configured `Origin` matching rules. The server is return a `200/OK` along with a `Access-Control-Allow-Origin` value that matches the specific `Origin` provided.

CORS Acceptance Response

```
$ curl -v http://127.0.0.1:8080/api/anonymous/hello?name=jim -H "Origin: http://localhost:8080"  
* Trying 127.0.0.1:8080...  
* Connected to 127.0.0.1 (127.0.0.1) port 8080 (#0)  
> GET /api/anonymous/hello?name=jim HTTP/1.1  
> Host: 127.0.0.1:8080 ①  
> Origin: http://localhost:8080 ②  
>  
< HTTP/1.1 200  
< Vary: Origin  
< Vary: Access-Control-Request-Method  
< Vary: Access-Control-Request-Headers  
< Access-Control-Allow-Origin: http://localhost:8080 ②  
hello, jim :caller=(null)
```

① Example Host and Origin have been flipped to match approved localhost:8080 Origin

② **Access-Control-Allow-Origin** denotes approval for the given Origin

202.6. CORS Server Rejection

This additional definition is enough to produce a **403/FORBIDDEN** from the server versus a rejection from the browser.

CORS Rejection Response

```
$ curl -v http://localhost:8080/api/anonymous/hello?name=jim -H "Origin: http://127.0.0.1:8080"  
> GET /api/anonymous/hello?name=jim HTTP/1.1  
> Host: localhost:8080  
> Origin: http://127.0.0.1:8080  
>  
< HTTP/1.1 403  
< Vary: Origin  
< Vary: Access-Control-Request-Method  
< Vary: Access-Control-Request-Headers  
Invalid CORS request
```

202.7. Spring MVC @CrossOrigin Annotation

Spring also offers an annotation-based way to enable the CORS protocol. In the example below, **@CrossOrigin** annotation has been added to the controller class or individual operations indicating CORS constraints.

This technique is static.

Spring MVC @CrossOrigin Annotation

```
...
import org.springframework.web.bind.annotation.CrossOrigin;
...
@CrossOrigin ①
@RestController
public class HelloController {
```

① defaults to all origins, etc.

Chapter 203. RestTemplate Authentication

Now that we have locked down our endpoints — requiring authentication — I want to briefly show how we can authenticate with `RestTemplate` using an existing BASIC Authentication filter. I am going to delay demonstrating `WebClient` to limit the dependencies on the current example application — but we will do so in a similar way that does not change the interface to the caller.

203.1. ClientHttpRequestFactory

We will first define a factory for creating client connections. It is quite simple here because we are not addressing things like HTTP/TLS connections. However, creating the bean external from the clients makes the clients connection agnostic.

ClientHttpRequestFactory

```
@Bean  
ClientHttpRequestFactory requestFactory() {  
    return new SimpleClientHttpRequestFactory();  
}
```



This simple `ClientHttpRequestFactory` will get slightly more complicated when we enable SSL connections. By instantiating it now in a separate method we will make the rest of the `RestTemplate` configuration oblivious to the SSL/non-SSL configuration.

203.2. Anonymous RestTemplate

The following snippet is an example of a `RestTemplate` representing an anonymous user. This should look familiar to what we have used prior to security.

RestTemplate Anonymous Client

```
@Bean  
public RestTemplate anonymousUser(RestTemplateBuilder builder,  
                                  ClientHttpRequestFactory requestFactory) {  
    RestTemplate restTemplate = builder.requestFactory(  
        //used to read the streams twice -- so we can use the logging filter below  
        ()->new BufferingClientHttpRequestFactory(requestFactory))  
        .interceptors(new RestTemplateLoggingFilter())  
        .build(); ①  
    return restTemplate;  
}
```

① vanilla RestTemplate with our debug log interceptor

203.3. Authenticated RestTemplate

The following snippet is an example of a RestTemplate that will authenticate as "user/password" using Http BASIC Authentication. The authentication is added as a filter along with the logger. The business code using this client will be ignorant of the extra authentication details.

RestTemplate Authenticating Client

```
@Bean
public RestTemplate authnUser(RestTemplateBuilder builder,
                               ClientHttpRequestFactory requestFactory) {
    RestTemplate restTemplate = builder.requestFactory(
        //used to read the streams twice -- so we can use the logging filter below
        ()->new BufferingClientHttpRequestFactory(requestFactory))
        .interceptors(new BasicAuthenticationInterceptor("user", "password"), ①
                      new RestTemplateLoggingFilter())
        .build();
    return restTemplate;
}
```

① added BASIC Auth filter to add Authorization Header

203.4. Authentication Integration Tests with RestTemplate

The following shows the different `RestTemplate` instances being injected that have different credentials assigned. The different attribute names, matching the `@Bean` factory names act as a qualifier to supply the right instance of `RestTemplate`.

```
@SpringBootTest(classes= ClientTestConfiguration.class,
                 webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT,
                 properties = "test=true") ①
public class AuthnRestTemplateNTest {
    @Autowired
    private RestTemplate anonymousUser;
    @Autowired
    private RestTemplate authnUser;
```

① test property triggers Swagger `@Configuration` and anything else not suitable during testing to disable

Chapter 204. RestClient Authentication

Let's also show how to authenticate with `RestClient`. We can do that using a builder or use a fully configured `RestTemplate`.

204.1. Anonymous RestClient

The following snippet is an example of a `RestClient` representing an anonymous user.

RestClient Anonymous Client

```
@Bean
public RestClient anonymousUserClient(RestClient.Builder builder,
ClientHttpRequestFactory requestFactory) {
    return builder.requestFactory("//used to read streams twice -- to use logging
filter
        new BufferingClientHttpRequestFactory(requestFactory))
    .requestInterceptor(new RestTemplateLoggingFilter())
    .build();
}
```

① vanilla `RestClient` with our debug log interceptor

204.2. Authenticated RestTemplate

The following snippet is an example of a `RestClient` built from an existing `RestTemplate`. It will also authenticate as "user/password" using Http BASIC Authentication.

RestClient Authenticating Client

```
@Bean
public RestClient authnUserClient(RestTemplate authnUser) {
    return RestClient.create(authnUser); ①
}
```

① uses already assembled filters from `RestTemplate`

Chapter 205. Mock MVC Authentication

There are many test frameworks within Spring and Spring Boot that I did not cover earlier. I limited them because covering them all early on added limited value with a lot of volume. However, I do want to show you a small example of MockMvc and how it can be configured for authentication. The following example shows a:

- normal injection of the mock that will be an anonymous user
- how to associate a mock to the security context

MockMvc Authentication Setup

```
@SpringBootTest(  
    properties = "test=true")  
@AutoConfigureMockMvc  
public class AuthConfigMockMvcNTest {  
    @Autowired  
    private WebApplicationContext context;  
    @Autowired  
    private MockMvc anonymous; //letting MockMvc do the setup work  
    private MockMvc userMock; //example manual instantiation ①  
    private final String uri = "/api/anonymous/hello";  
  
    @BeforeEach  
    public void init() {  
        userMock = MockMvcBuilders //the rest of manual instantiation  
            .webAppContextSetup(context)  
            .apply(SecurityMockMvcConfigurers.springSecurity())  
            .build();  
    }  
}
```

① there is no functional difference between the injected or manually instantiated `MockMvc` the way it is performed here

205.1. MockMvc Anonymous Call

The first test is a baseline example showing a call through the mock to a service that allows all callers and no required authentication. The name of the mock is not important. It is an anonymous client at this point because we have not assigned it any identity.

MockMvc Anonymous Call

```
@Test  
public void anonymous_can_call_get() throws Exception {  
    anonymous.perform(MockMvcRequestBuilders.get(uri).queryParam("name", "jim"))  
        .andDo(print())  
        .andExpect(status().isOk())  
        .andExpect(content().string("hello, jim :caller=(null)"));  
}
```

```
}
```

205.2. MockMvc Authenticated Call

The next example shows how we can inject an identity into the mock for use during the test method. We can use an injected or manual mock for this. The important point to notice is that the mock user's identity is assigned through an annotation on the `@Test`.

MockMvc Authenticated Call

```
@WithMockUser("user")
@Test
public void user_can_call_get() throws Exception {
    userMock.perform(MockMvcRequestBuilders.get(uri)
        .queryParam("name","jim"))
        .andDo(print())
        .andExpect(status().isOk())
        .andExpect(content().string("hello, jim :caller=user"));
}
```

Although I believe RestTemplate tests are pretty good at testing client access—the WebMvc framework was a very convenient to quickly verify and identify issues with the `SecurityFilterChain` definitions.

205.3. MockMvc does not require SpringBootTest

The MockMvc web test framework does not require the full application context implemented by `SpringBootTest`. MockMvc provides a means to instantiate small unit tests incorporating mocks behind the controllers. For example, I have used it as a lightweight way to test `ControllerAdvice`/`ExceptionAdvice`.

Chapter 206. Summary

In this module, we learned:

- how to configure a `SecurityFilterChain`
- how to define no security filters for static resources
- how to customize the `SecurityFilterChain` for API endpoints
- how to expose endpoints that can be called from anonymous users
- how to require authenticated users for certain endpoints
- how to CORS-enable the API
- how to define BASIC Auth for OpenAPI and for use by Swagger
- how to add identity to RestTemplate and RestClient clients

User Details

copyright © 2024 jim stafford (jim.stafford@jhu.edu)

Chapter 207. Introduction

In previous sections we looked closely at how to authenticate a user obtained from a demonstration user source. The focus was on the obtained user and the processing that went on around it to enforce authentication using an example credential mechanism. There was a lot to explore with just a single user relative to establishing the security filter chain, requiring authentication, supplying credentials with the call, completing the authentication, and obtaining the authenticated user identity.

In this chapter we will focus on the `UserDetailsService` framework that supports the `AuthenticationProvider` so that we can implement multiple users, multiple user information sources, and to begin storing those users in a database.

207.1. Goals

You will learn:

- the interface roles in authenticating users within Spring
- how to configure authentication and authentication sources for use by a security filter chain
- how to implement access to user details from different sources
- how to implement access to user details using a database

207.2. Objectives

At the conclusion of this lecture and related exercises, you will be able to:

1. build various `UserDetailsService` implementations to host user accounts and be used as a source for authenticating users
2. build a simple in-memory `UserDetailsService`
3. build an injectable `UserDetailsService`
4. build a `UserDetailsService` using access to a relational database
5. configure an application to display the database UI
6. encode passwords

Chapter 208. AuthenticationManager

The focus of this chapter is on providing authentication to stored users and providing details about them. To add some context to this, lets begin the presentation flow with the [AuthenticationManager](#).

[AuthenticationManager](#) is an abstraction the code base looks for in order to authenticate a set of credentials. Its input and output are of the same interface type—[Authentication](#)—but populated differently and potentially implemented differently.

The input [Authentication](#) primarily supplies the principal (e.g., username) and credentials (e.g., plaintext password). The output [Authentication](#) of a successful authentication supplies resolved [UserDetails](#) and provides direct access to granted authorities—which can come from those user details and will be used during later authorizations. Although the credentials (e.g., encrypted password hash) from the stored [UserDetails](#) is used to authenticate, it's contents are cleared before returning the response to the caller.



Figure 88. `AuthenticationManager` and `UserDetails`

208.1. ProviderManager

The `AuthenticationManager` is primarily implemented using the `ProviderManager` class and delegates authentication to its assigned `AuthenticationProviders` and/or parent `AuthenticationManager` to do the actual authentication. Some `AuthenticationProvider` classes are based off a `UserDetailsService` to provide `UserDetails`. However, that is not always the case—therefore the diagram below does not show a direct relationship between the `AuthenticationProvider` and `UserDetailsService`.

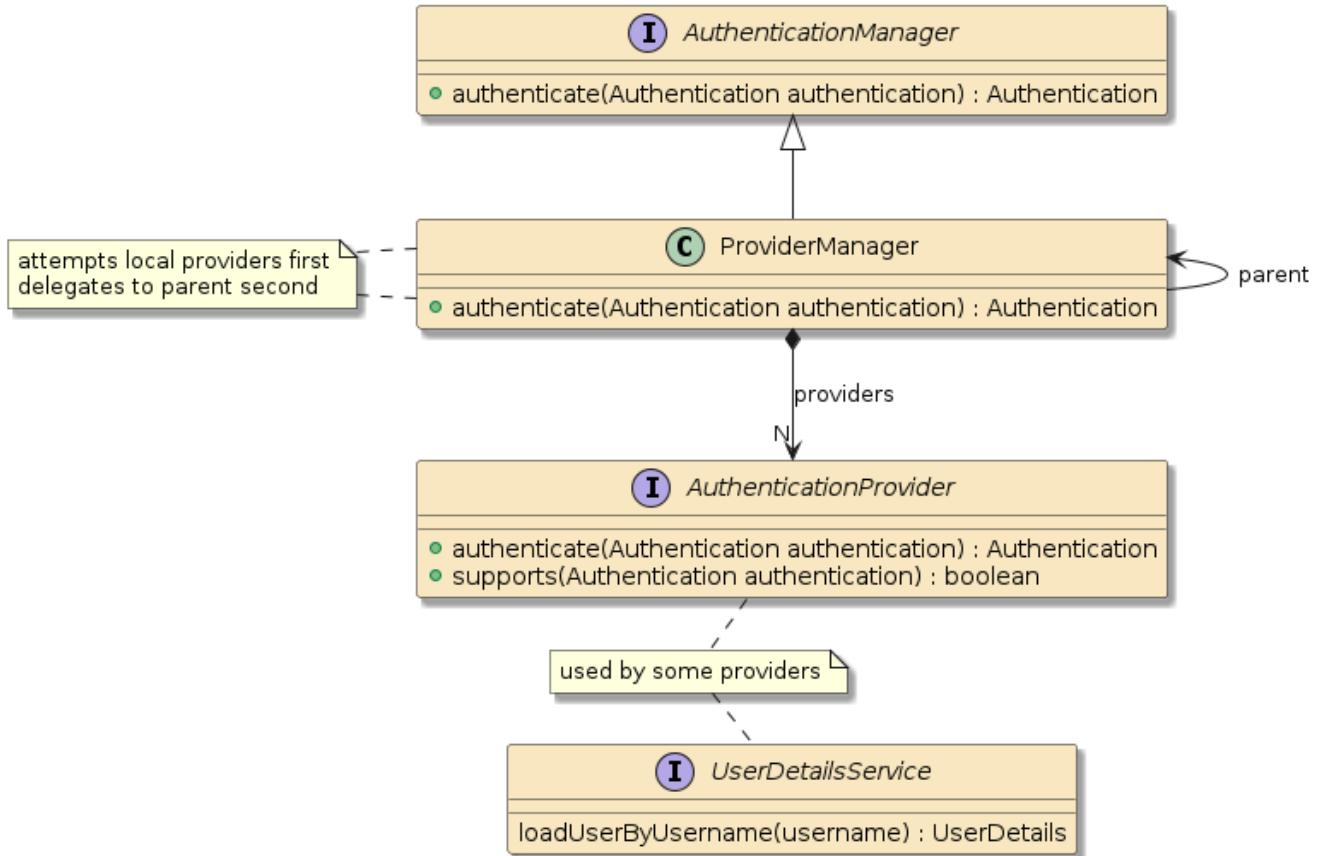


Figure 89. ProviderManager

208.2. AuthenticationManagerBuilder

It is the job of the `AuthenticationManagerBuilder` to assemble an `AuthenticationManager` with the required `AuthenticationProviders` and—where appropriate—`UserDetailsService`. The `AuthenticationManagerBuilder` is configured during the assembly of the `SecurityFilterChain` in both the legacy WebSecurityConfigurerAdapter and modern Component-based approaches.

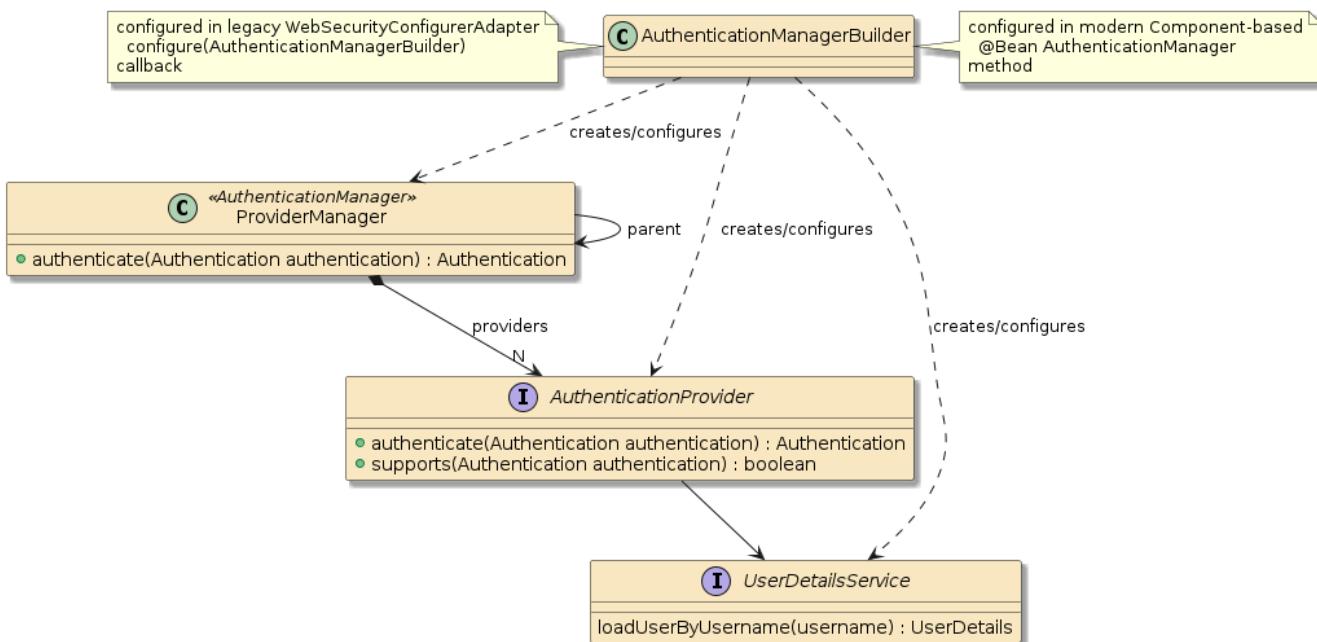


Figure 90. AuthenticationManagerBuilder

One can custom-configure the `AuthenticationProviders` for the `AuthenticationManagerBuilder` in the legacy `WebSecurityConfigurerAdapter` approach by overriding the `configure()` callback.

Configure AuthenticationManagerBuilder using legacy WebSecurityConfigurerAdapter

```
@Configuration(proxyBeanMethods = false)
public static class APIConfiguration extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        ... ①
    }
}
```

① can custom-configure `AuthenticationManagerBuilder` here during a `configure()` callback

One can custom-configure the `AuthenticationProviders` for the `AuthenticationManagerBuilder` in the modern Component-based approach by obtaining it from an injected `HttpSecurity` object using the `getSharedObject()` call.

Configure AuthenticationManagerBuilder using modern Component-based Approach

```
@Bean
public AuthenticationManager authnManager(HttpSecurity http, ...) throws Exception {
    AuthenticationManagerBuilder builder =
        http.getSharedObject(AuthenticationManagerBuilder.class);
    ...①

    builder.parentAuthenticationManager(null); //prevent from being recursive ②
    return builder.build();
}
```

① can obtain and custom-configure `AuthenticationManagerBuilder` using injected `HttpSecurity` object

② I found the need to explicitly define "no parent" in the modern Component-based approach

208.3. AuthenticationManagerBuilder Builder Methods

We can use the local builder methods to custom-configure the `AuthenticationManagerBuilder`. These allow us to assemble one or more of the well-known `AuthenticationProvider` types. The following is an example of configuring an `InMemoryUserDetailsManager` that our earlier examples used in the previous chapters. However, in this case we get a chance to explicitly populate with users.



This is an early example demonstration toy

Example InMemoryAuthentication Configuration

```
PasswordEncoder encoder = ...
builder.inMemoryAuthentication() ①
    .passwordEncoder(encoder) ②
    .withUser("user1").password(encoder.encode("password1")).roles() ③
```

```
.and()
.withUser("user2").password(encoder.encode("password1")).roles();
```

- ① adds a `UserDetailsService` to `AuthenticationManager` implemented in memory
- ② `AuthenticationProvider` will need a password encoder to match passwords during authentication
- ③ users placed directly into storage must have encoded password

208.3.1. Assembled AuthenticationProvider

The results of the builder configuration are shown below where the builder assembled an `AuthenticationManager` (`ProviderManager`) and populated it with an `AuthenticationProvider` (`DaoAuthenticationProvider`) that can work with the `UserDetailsService` (`InMemoryUserDetailsManager`) we identified.

The builder also populated the `UserDetailsService` with two users: `user1` and `user2` with an encoded password using the `PasswordEncoder` also set on the `AuthenticationProvider`.

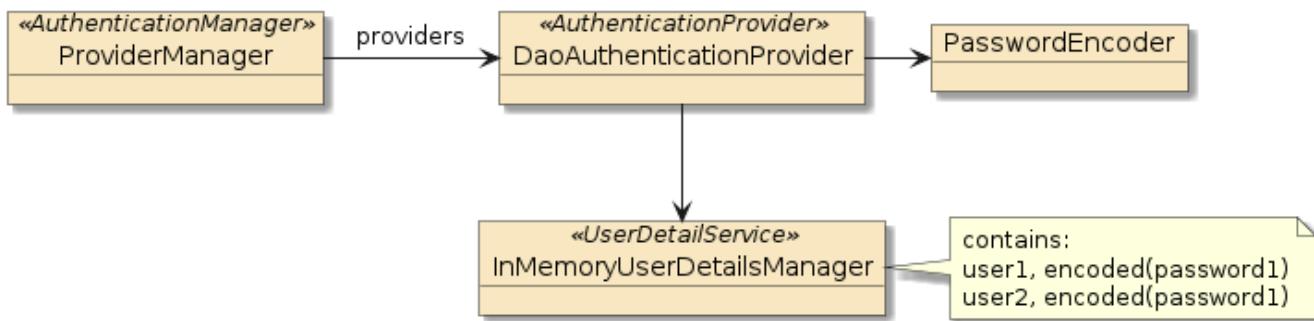


Figure 91. Example `InMemoryUserDetailsManager`

208.3.2. Builder Authentication Example

With that in place—we can authenticate our two users using the `UserDetailsService` defined and populated using the builder.

Builder Authentication Example

```
$ curl http://localhost:8080/api/authn/hello?name=jim -u user1:password1
hello, jim :caller=user1

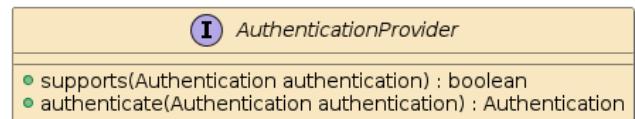
$ curl http://localhost:8080/api/authn/hello?name=jim -u user2:password1
hello, jim :caller=user2

$ curl http://localhost:8080/api/authn/hello?name=jim -u userX:password -v
< HTTP/1.1 403
```

208.4. AuthenticationProvider

The `AuthenticationProvider` can answer two (2) questions:

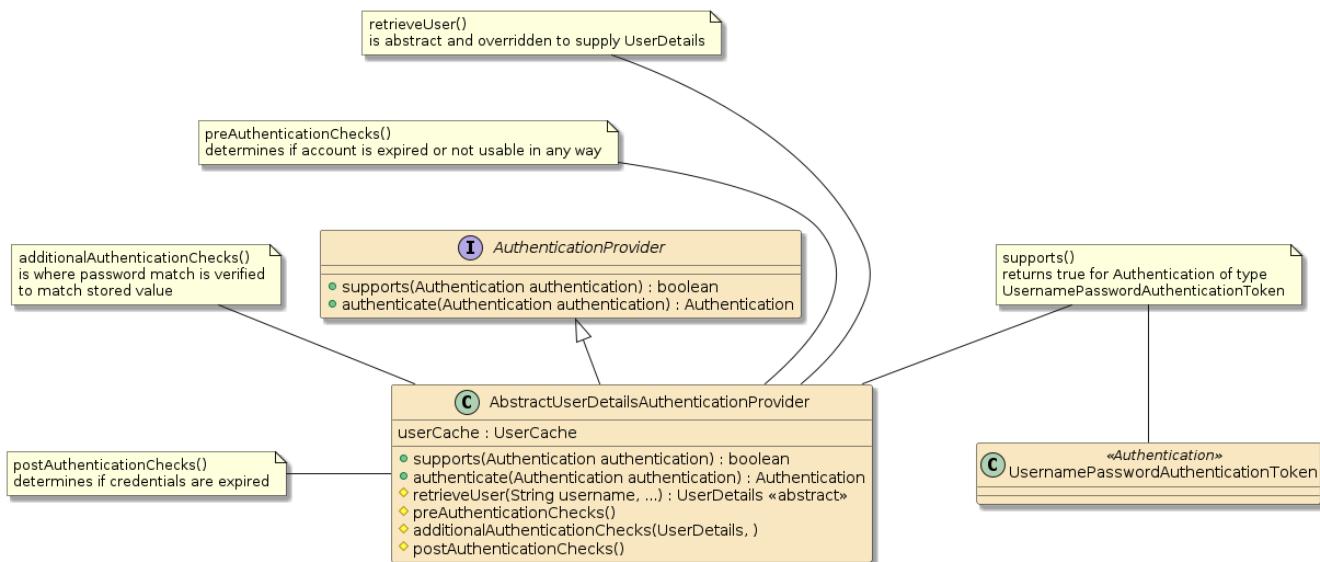
- do you support this type of authentication
- can you authenticate this attempt



208.5. AbstractUserDetailsAuthenticationProvider

For username/password authentication, Spring provides an **AbstractUserDetailsAuthenticationProvider** that supplies the core authentication workflow that includes:

- a **UserCache** to store **UserDetails** from previous successful lookups
- obtaining the **UserDetails** if not already in the cache
- pre and post-authorization checks to verify such things as the account locked/disabled/expired or the credentials expired.
- additional authentication checks where the password matching occurs



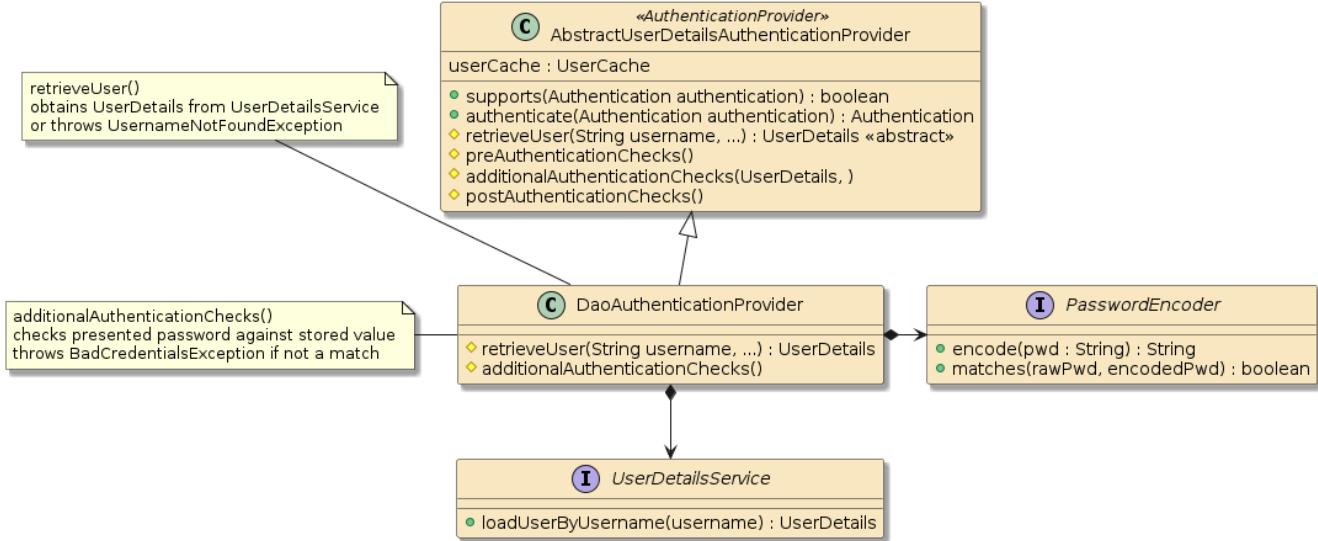
The instance will support any authentication token of type **UsernamePasswordAuthenticationToken** but will need at least two things:

- user details from storage
- a means to authenticate presented password

208.6. DaoAuthenticationProvider

Spring provides a concrete **DaoAuthenticationProvider** extension of the **AbstractUserDetailsAuthenticationProvider** class that works directly with:

- **UserDetailsService** to obtain the **UserDetails**
- **PasswordEncoder** to perform password matching

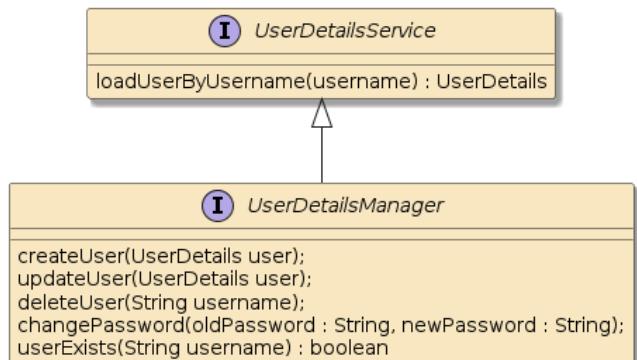


Now all we need is a **PasswordEncoder** and **UserDetailsService** to get all this rolling.

208.7. UserDetailsManager

Before we get too much further into the details of the **UserDetailsService**, it will be good to be reminded that the interface supplies only a single `loadUserByUsername()` method.

There is an extension of that interface to address full lifecycle **UserDetails** management and some of the implementations I will reference implement one or both of those interfaces. We will, however, focus only on the authentication portion and ignore most of the other lifecycle aspects for now.



Chapter 209. AuthenticationManagerBuilder Configuration

At this point we know the framework of objects that need to be in place for authentication to complete and how to build a toy `InMemoryUserDetailsManager` using builder methods within the `AuthenticationManagerBuilder` class.

In this section we will learn how we can configure additional sources with less assistance from the `AuthenticationManagerBuilder`.

209.1. Fully-Assembled AuthenticationManager

We can directly assign a fully-assembled `AuthenticationManager` to other `SecurityFilterChains` by first exporting it as a `@Bean`.

- The legacy `WebSecurityConfigurerAdapter` approach provides a `authenticationManagerBean()` helper method that can be exposed as a `@Bean` by the derived class.

`@Bean AuthenticationManager — legacy WebSecurityConfigurerAdapter approach`

```
@Configuration
public class APIConfiguration extends legacy WebSecurityConfigurerAdapter {
    @Bean
    @Override
    public AuthenticationManager authenticationManagerBean() throws Exception {
        return super.authenticationManagerBean();
    }
}
```

- The modern Component-based approach builds an `AuthenticationManager` in a `@Bean` factory method using a builder accessible from the injected `HttpSecurity`.

`@Bean AuthenticationManager — modern Component-based approach`

```
@Bean
public AuthenticationManager authnManager(HttpSecurity http,...) throws Exception {
    AuthenticationManagerBuilder builder =
        http.getSharedObject(AuthenticationManagerBuilder.class);
    ...
    builder.parentAuthenticationManager(null); //prevent from being recursive
    return builder.build();
}
```

With the fully-configured `AuthenticationManager` exposed as a `@Bean`, we can look to directly wire it into the other `SecurityFilterChains`.

209.2. Directly Wire-up AuthenticationManager

We can directly set the `AuthenticationManager` to one created elsewhere. The following examples shows setting the `AuthenticationManager` during the building of the `SecurityFilterChain`.

- legacy `WebSecurityConfigurerAdapter` approach

Assigning Parent AuthenticationManager — legacy WebSecurityConfigurerAdapter Approach

```
@Configuration
@Order(500)
@RequiredArgsConstructor
public static class H2Configuration extends WebSecurityConfigurerAdapter {
    private final AuthenticationManager authenticationManager; ①

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.requestMatchers(m->m.antMatchers(...));
        ...
        http.authenticationManager(authenticationManager); ②
    }
}
```

① `AuthenticationManager` assembled elsewhere and injected in this `@Configuration` class

② injected `AuthenticationManager` to be the `AuthenticationManager` for what this builder builds

- modern Component-based approach

Assigning AuthenticationManager — modern Component-based Approach

```
@Order(500)
@Bean
public SecurityFilterChain h2SecurityFilters(HttpSecurity http, ①
                                             AuthenticationManager authMgr) throws
Exception {
    http.securityMatchers(cfg->cfg.requestMatchers(...));
    ...
    http.authenticationManager(authMgr); ②
    return http.build();
}
```

① `AuthenticationManager` assembled elsewhere and injected in this `@Bean` factory method

② injected `AuthenticationManager` to be the `AuthenticationManager` for what this builder builds

209.3. Directly Wire-up Parent AuthenticationManager

We can instead set the parent `AuthenticationManager` using the `SecurityAuthenticationManagerBuilder`.

- The following example shows setting the parent `AuthenticationManager` during a `WebSecurityConfigurerAdapter.configure()` callback in the legacy WebSecurityConfigurerAdapter approach.

Assigning Parent AuthenticationManager—legacy WebSecurityConfigurerAdapter Approach

```
@Configuration
@Order(500)
@RequiredArgsConstructor
public static class H2Configuration extends WebSecurityConfigurerAdapter {
    private final AuthenticationManager authenticationManager;

    @Override
    protected void configure(AuthenticationManagerBuilder auth)
        throws Exception {
        auth.parentAuthenticationManager(authenticationManager); ①
    }
}
```

① injected `AuthenticationManager` to be the parent `AuthenticationManager` of what this builder builds

- The following example shows setting the parent `AuthenticationManager` during the build of the `SecurityFilterChain` using `http.getSharedObject()`.

Assigning Parent AuthenticationManager—modern Component-based Approach

```
@Order(500)
@Bean
public SecurityFilterChain h2SecurityFilters(HttpSecurity http,
                                             AuthenticationManager authMgr) throws
Exception {
    ...
    AuthenticationManagerBuilder builder =
        http.getSharedObject(AuthenticationManagerBuilder.class);
    builder.parentAuthenticationManager(authMgr); ①
    return http.build();
}
```

① injected `AuthenticationManager` to be the parent `AuthenticationManager` of what this builder builds

209.4. Define Service and Encoder @Bean

Another option in supplying a `UserDetailsService` is to define a globally accessible `UserDetailsService @Bean` to inject to use with our builder. However, in order to pre-populate the `UserDetails` passwords, we must use a `PasswordEncoder` that is consistent with the `AuthenticationProvider` this `UserDetailsService` will be combined with. We can set the default `PasswordEncoder` using a `@Bean` factory.

Defining a Default PasswordEncoder for AuthenticationProvider

```
@Bean ①
public PasswordEncoder passwordEncoder() {
    return ...
}
```

- ① defining a `PasswordEncoder` to be injected into default `AuthenticationProvider`

Defining Injectable UserDetailsService

```
@Bean
public UserDetailsService sharedUserDetailsService(PasswordEncoder encoder) { ①
    //caution -- authorities accumulate in 3.1.0 / Spring 6.1.1, fixed in 6.2.0-M1
    //create new builder for each new user to avoid issue
    List<UserDetails> users = List.of(
        User.withUsername("user1").passwordEncoder(encoder::encode) ②
            .password("password2").roles().build(), ③
        User.withUsername("user3").passwordEncoder(encoder::encode)
            .password("password2").roles().build()
    );
    return new InMemoryUserDetailsManager(users);
}
```

- ① using an injected `PasswordEncoder` for consistency
② using different `UserDetails` builder than before — setting password encoding function
③ username user1 will be in both `UserDetailsService` with different passwords



`User.withUsername()` constructs new `User` builder.

209.4.1. Inject UserDetailService

We can inject the fully-assembled `UserDetailsService` into the `AuthenticationManagerBuilder` — just like before with the `inMemoryAuthentication`, except this time the builder has no knowledge of the implementation being injected. We are simply injecting a `UserDetailsService`. The builder will accept it and wrap that in an `AuthenticationProvider`

Inject Fully-Assembled UserDetails Service — legacy WebSecurityConfigurerAdapter Approach

```
@Configuration
@Order(0)
@RequiredArgsConstructor
public static class APIConfiguration extends WebSecurityConfigurerAdapter {
    private final List<UserDetailsService> userDetailsServicees; ①

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        ...
        for (UserDetailsService uds: userDetailsServicees) {
```

```

        auth.userDetailsService(uds); ②
    }
}

```

① injecting `UserDetailsService` into configuration class

② adding additional `UserDetailsService` to create additional `AuthenticationProvider`

The same can be done in the modern Component-based approach and during the equivalent builder configuration I demonstrated earlier with the `inMemoryAuthentication`. The only difference is that I found the more I custom-configured the `AuthenticationManagerBuilder`, I would end up in a circular configuration with the `AuthenticationManager` pointing to itself as its parent unless I explicitly set the parent value to null.

Inject Fully-Assembled UserDetailsService — modern Component-based Approach

```

@Bean
public AuthenticationManager authnManager(HttpSecurity http,
    List<UserDetailsService> userDetailsServicees ) throws Exception { ①
    AuthenticationManagerBuilder builder = http.getSharedObject
(AuthenticationManagerBuilder.class);
    ...
    for (UserDetailsService uds : userDetailsServicees) {
        builder.userDetailsService(uds); ②
    }

    builder.parentAuthenticationManager(null); //prevent from being recursive
    return builder.build();
}

```

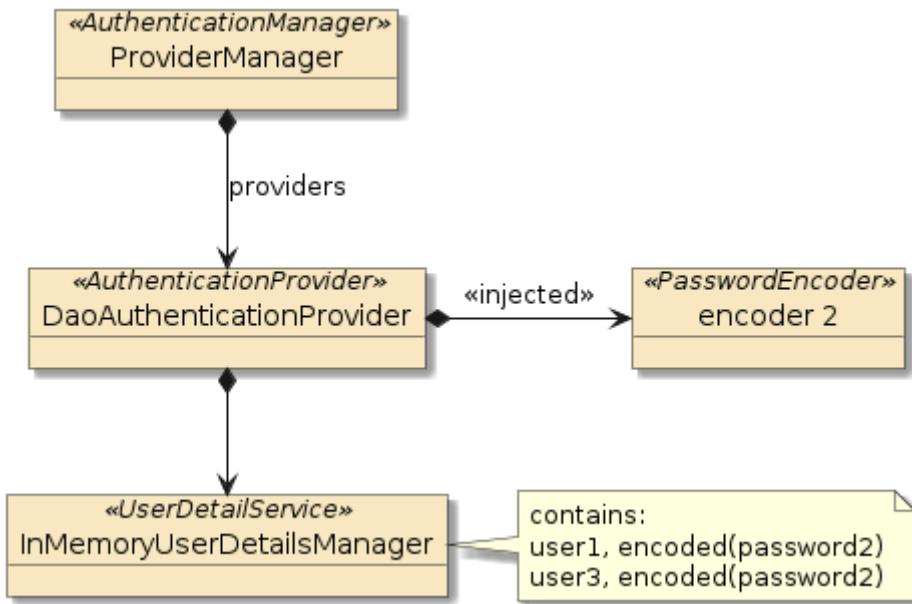
① injecting `UserDetailsService` into bean method

② adding additional `UserDetailsService` to create additional `AuthenticationProvider`

209.4.2. Assembled Injected UserDetailsService

The results of the builder configuration are shown below where the builder assembled an `AuthenticationProvider` (`DaoAuthenticationProvider`) based on the injected `UserDetailsService` (`InMemoryUserDetailsService`).

The injected `UserDetailsService` also had two users—`user1` and `user3`—added with an encoded password based on the injected `PasswordEncoder` bean. This will be the same bean injected into the `AuthenticationProvider`.



209.4.3. Injected UserDetailsService Example

With that in place, we can now authenticate `user1` and `user3` using the assigned passwords using the `AuthenticationProvider` with the injected `UserDetailsService`.

Injected UserDetailsService Authentication Example

```

$ curl http://localhost:8080/api/authn/hello?name=jim -u user1:password2
hello, jim :caller=user1

$ curl http://localhost:8080/api/authn/hello?name=jim -u user3:password2
hello, jim :caller=user3

$ curl http://localhost:8080/api/authn/hello?name=jim -u userX:password -v
< HTTP/1.1 403

```

209.5. Combine Approaches

As stated before—the `ProviderManager` can delegate to multiple `AuthenticationProviders` before authenticating or rejecting an authentication request. We have demonstrated how to create an `AuthenticationManager` multiple ways. In this example, I am integrating the two `AuthenticationProviders` into a single `AuthenticationManager`.

Defining Multiple AuthenticationProviders

```

//AuthenticationManagerBuilder auth
PasswordEncoder encoder = ... ①
auth.inMemoryAuthentication().passwordEncoder(encoder)
    .withUser("user1").password(encoder.encode("password1")).roles()
    .and()
    .withUser("user2").password(encoder.encode("password1")).roles();
for (UserDetailsService uds : userDetailsServicees) { ②
    builder.userDetailsService(uds);
}

```

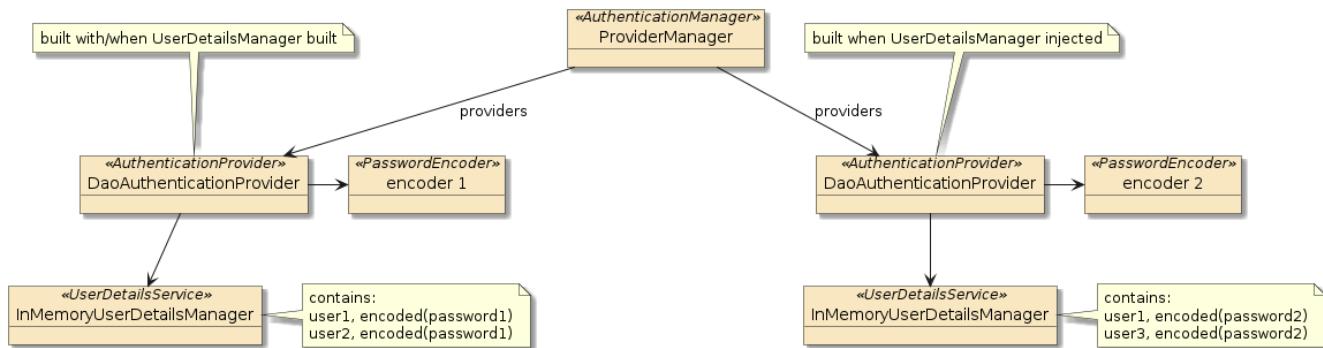
```
}
```

- ① locally built `AuthenticationProvider` will use its own encoder
- ② `@Bean`-built `UserDetailsService` injected and used to form second `AuthenticationProvider`

209.5.1. Assembled Combined AuthenticationProviders

The resulting `AuthenticationManager` ends up with two custom-configured `AuthenticationProviders`. Each `AuthenticationProviders` are

- implemented with the `DaoAuthenticationProvider` class
- make use of a `PasswordEncoder` and `UserDetailsService`



The left `UserDetailsService` was instantiated locally as an `InMemoryUserDetailsService`, using a `@Bean` factory that instantiated the builder methods from the `InMemoryUserDetailsService` directly. Since it was `AuthenticationManagerBuilder`. Since it was shared as a `@Bean`, the factory method used an locally built, it was building the injected `PasswordEncoder` to assemble. `AuthenticationProvider` at the same time and could define its own choice of `PasswordEncoder`

The two were brought together by one of our configuration approaches and now we have two sources of credentials to authenticate against.

209.5.2. Multiple Provider Authentication Example

With the two `AuthenticationProvider` objects defined, we can now login as user2 and user3, and user1 using both passwords. The user1 example shows that an authentication failure from one provider still allows it to be inspected by follow-on providers.

Multiple Provider Authenticate Example

```
$ curl http://localhost:8080/api/authn/hello?name=jim -u user1:password1
hello, jim :caller=user1

$ curl http://localhost:8080/api/authn/hello?name=jim -u user1:password2
hello, jim :caller=user1

$ curl http://localhost:8080/api/authn/hello?name=jim -u user2:password1
```

```
hello, jim :caller=user2
```

```
$ curl http://localhost:8080/api/authn/hello?name=jim -u user3:password2
hello, jim :caller=user3
```

Chapter 210. UserDetails

So now we know that all we need is to provide a `UserDetailsService` instance and Spring will take care of most of the rest. `UserDetails` is an interface that we can implement any way we want. For example—if we manage our credentials in MongoDB or use Java Persistence API (JPA), we can create the proper classes for that mapping. We won't need to do that just yet because Spring provides a `User` class that can work for most POJO-based storage solutions.

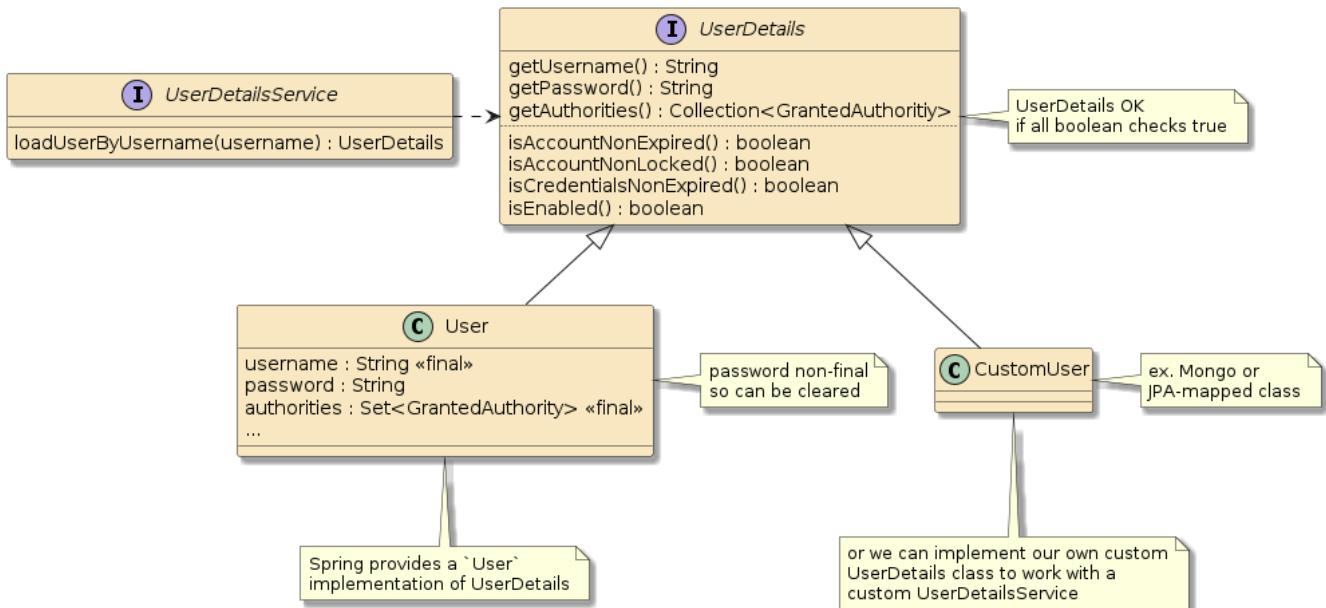


Figure 92. `UserDetailsService`

Chapter 211. PasswordEncoder

I have made mention several times about the [PasswordEncoder](#) and earlier covered how it is used to create a cryptographic hash. Whenever we configured a [PasswordEncoder](#) for our [AuthenticationProvider](#) we have the choice of many encoders. I will highlight three of them.

211.1. NoOpPasswordEncoder

The [NoOpPasswordEncoder](#) is what it sounds like. It does nothing when encoding the plaintext password. This can be used for early development and debug but should not—obviously—be used with real credentials.

211.2. BCryptPasswordEncoder

The [BCryptPasswordEncoder](#) uses a very strong Bcrypt algorithm and (the algorithm) likely should be considered the default in production environments.

211.3. DelegatingPasswordEncoder

The [DelegatingPasswordEncoder](#) is a jack-of-all-encoders. It has one default way to encode—using BCrypt—but can validate passwords of numerous algorithms. This encoder writes and relies on all passwords starting with an [{encoding-key}](#) that indicates the type of encoding to use.

Example DelegatingPasswordEncoder Values

```
{noop}password  
{bcrypt}$2y$10$UvKwrln7xPp35c5sbj.9kuZ9jY9VYg/VylVTu88ZSCYy/YdcdP/Bq
```

Use the [PasswordEncoderFactories](#) class to create a [DelegatingPasswordEncoder](#) populated with a full compliment of encoders.

Example Fully Populated DelegatingPasswordEncoder Creation

```
import org.springframework.security.crypto.factory.PasswordEncoderFactories;  
  
 @Bean  
 public PasswordEncoder passwordEncoder() {  
     return PasswordEncoderFactories.createDelegatingPasswordEncoder();  
 }
```

DelegatingPasswordEncoder encodes one way and matches multiple ways

 [DelegatingPasswordEncoder](#) encodes using a single, designated encoder and matches against passwords encoded using many alternate encodings—thus relying on the password to start with a [{encoding-key}](#).

Chapter 212. JDBC UserDetailsService

Spring provides two Java Database Connectivity (JDBC) implementation classes that we can easily use out of the box to begin storing `UserDetails` in a database:

- `JdbcDaoImpl` - implements just the core `UserDetailsService loadUserByUsername` capability
- `JdbcUserDetailManager` - implements the full `UserDetailsManager` CRUD capability

JDBC is a database communications interface containing no built-in mapping



JDBC is a low-level interface to access a relational database from Java. All the mapping between the database inputs/outputs and our Java business objects is done outside of JDBC. There is no mapping framework like with Java Persistence API (JPA).

`JdbcUserDetailManager` extends `JdbcDaoImpl`. We only need `JdbcDaoImpl` since we will only be performing authentication reads and not yet be implementing full CRUD (Create, Read, Update, and Delete) with databases. However, there would have been no harm in using the full `JdbcUserDetailManager` implementation in the examples below and simply ignored the additional behavior.

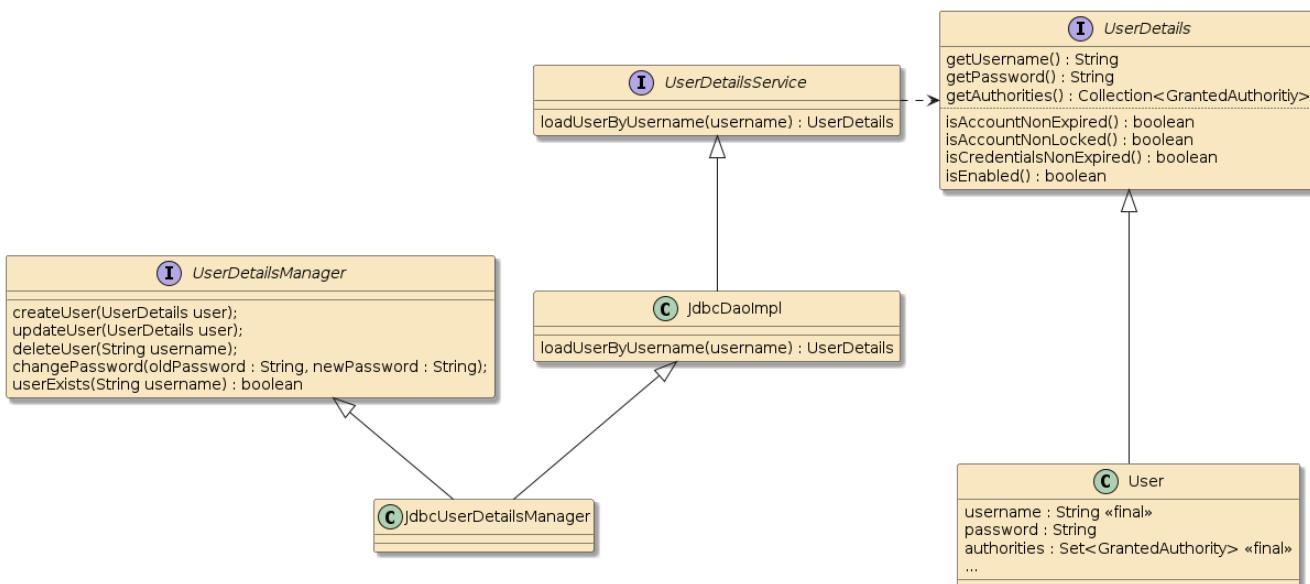


Figure 93. JDBC UserDetailsService

To use the JDBC implementation, we are going to need a few things:

- A relational database - this is where we will store our users
- Database Schema - this defines the tables and columns of the database
- Database Contents - this defines our users and passwords
- `javax.sql.DataSource` - this is a JDBC wrapper around a connection to the database
- construct the `UserDetailsService` (and potentially expose as a `@Bean`)
- inject and add JDBC `UserDetailsService` to `AuthenticationManagerBuilder`

212.1. H2 Database

There are [several lightweight databases](#) that are very good for development and demonstration (e.g., [h2](#), [hsqldb](#), [derby](#), [SQLite](#)). They commonly offer in-memory, file-based, and server-based instances with minimal scale capability but extremely simple to administer. In general, they supply an interface that is compatible with the more enterprise-level solutions that are more suitable for production. That makes them an ideal choice for using in demonstration and development situations like this. For this example, I will be using the [h2](#) database but many others could have been used as well.

212.2. DataSource: Maven Dependencies

To easily create a default DataSource, we can simply add a compile dependency on [spring-boot-starter-data-jdbc](#) and a runtime dependency on the [h2](#) database. This will cause our application to start with a default DataSource connected to the an in-memory database.

DataSource Maven Dependency

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jdbc</artifactId>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

212.3. JDBC UserDetailsService

Once we have the [spring-boot-starter-data-jdbc](#) and database dependency in place, Spring Boot will automatically create a default [javax.sql.DataSource](#) that can be injected into a [@Bean](#) factory so that we can create a [JdbcDaoImpl](#) to implement the JDBC [UserDetailsService](#).

JDBC UserDetailsService

```
import javax.sql.DataSource;
...
@Bean
public UserDetailsService jdbcUserDetailsService(DataSource userDataSource) {
    JdbcDaoImpl jdbcUds = new JdbcDaoImpl();
    jdbcUds.setDataSource(userDataSource);
    return jdbcUds;
}
```

From there, we can inject the JDBC [UserDetailsService](#)—like the in-memory version we injected earlier and add it to the builder.

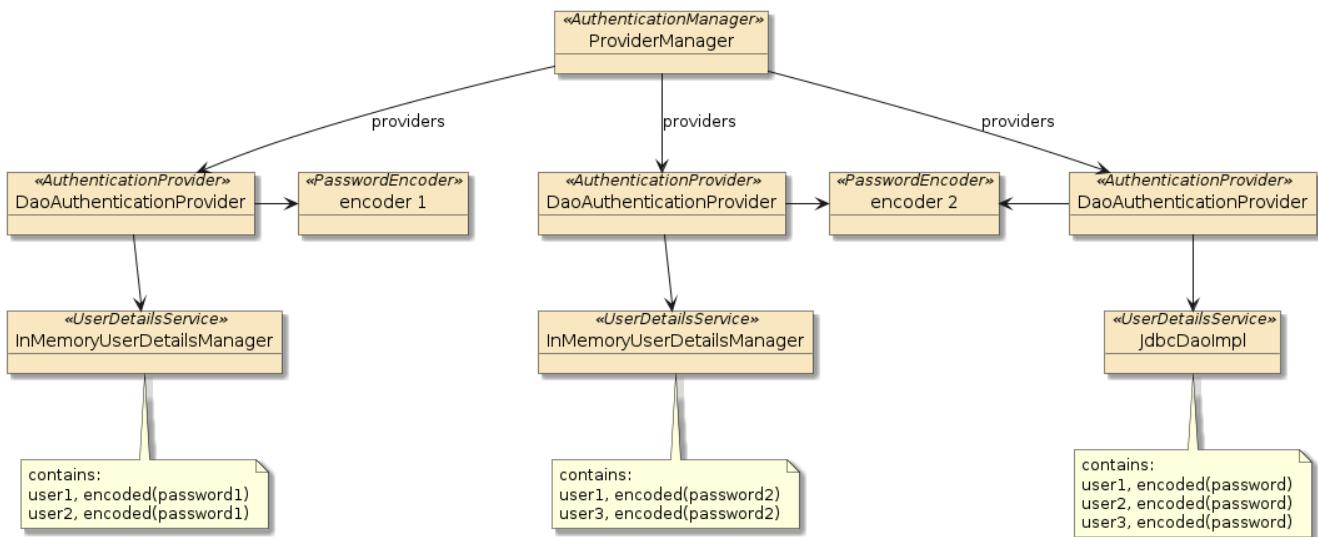


Figure 94. Aggregate Set of `UserDetailsServicees` and `AuthenticationProviders`

212.4. Autogenerated Database URL

If we restart our application at this point, we will get a generated database URL using a UUID for the name.

Autogenerated Database URL Output

```
H2 console available at '/h2-console'. Database available at  
'jdbc:h2:mem:76567045-619b-4588-ae32-9154ba9ac01c'
```

212.5. Specified Database URL

We can make the URL more stable and well-known by setting the `spring.datasource.url` property.

Setting DataSource URL

```
spring.datasource.url=jdbc:h2:mem:users
```

Specified Database URL Output

```
H2 console available at '/h2-console'. Database available at 'jdbc:h2:mem:users'
```



h2-console URI can be modified

We can also control the URI for the h2-console by setting the `spring.h2.console.path` property.

212.6. Enable H2 Console Security Settings

The h2 database can be used headless, but also comes with a convenient UI that will allow us to inspect the data in the database and manipulate it if necessary. However, with security

enabled—we will not be able to access our console by default. We only addressed authentication for the API endpoints. Since this is a chapter focused on configuring authentication, it is a good exercise to go through the steps to make the h2 UI accessible but also protected. The following will:

- require users accessing the `/h2-console/**` URIs to be authenticated
- enable FORM authentication and redirect successful logins to the `/h2-console` URI
- disable frame headers that would have placed constraints on how the console could be displayed
- disable CSRF for the `/h2-console/**` URI but leave it enabled for the other URIs
- wire in the injected `AuthenticationManager` configured for the API

212.6.1. H2 Configuration - legacy WebSecurityConfigurerAdapter Approach

H2 UI Security Configuration - legacy WebSecurityConfigurerAdapter Approach

```
@Configuration
@Order(500)
@RequiredArgsConstructor
public static class H2Configuration extends WebSecurityConfigurerAdapter {
    private final AuthenticationManager authenticationManager; ①

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.requestMatchers(m->m.antMatchers("/login", "/logout", "/h2-console/**"));
        http.authorizeRequests(cfg->cfg.antMatchers("/login", "/logout").permitAll());
    ②
        http.authorizeRequests(cfg->cfg.antMatchers("/h2-console/**").authenticated());
    ③
        http.csrf(cfg->cfg.ignoringAntMatchers("/h2-console/**")); ④
        http.headers(cfg->cfg.frameOptions().disable()); ⑤
        http.formLogin().successForwardUrl("/h2-console"); ⑥
        http.authenticationManager(authenticationManager); ⑦
    }
}
```

① injected `AuthenticationManager` bean exposed by `APIConfiguration`

② apply filter rules to H2 UI URIs as well as login/logout form

③ require authenticated users by the application to reach the console

④ turn off CSRF only for the H2 console

⑤ turn off display constraints for the H2 console

⑥ route successful logins to the H2 console

⑦ use pre-configured `AuthenticationManager` for authentication to UI

212.6.2. H2 Configuration—modern Component-based Approach

The following modern Component-based configuration addresses the URI builder changes in Spring Security 6. I needed to constrain the use of `/error` to the URI and request for `text/html` so that it did not interfere with the API errors. The use of the non-SpringMVC H2DB UI triggered the mandatory use of explicit URI builder calls to resolve ambiguous matching requirements.

H2 UI Configuration—modern Component-based Approach

```
import static
org.springframework.security.web.util.matcher.RegexRequestMatcher.regexMatcher;
import org.springframework.security.web.util.matcher.MediaTypeRequestMatcher;

@Order(500)
@Bean
public SecurityFilterChain h2SecurityFilters(HttpSecurity http,
    AuthenticationManager authMgr) throws Exception {
    MediaTypeRequestMatcher htmlRequestMatcher =
        new MediaTypeRequestMatcher(MediaType.TEXT_HTML);
    htmlRequestMatcher.setUseEquals(true);

    http.securityMatchers(cfg->cfg
        .requestMatchers("/h2-console*", "/h2-console/**")
        .requestMatchers("/login", "/logout")
        .requestMatchers(RequestMatchers.allOf(
            htmlRequestMatcher, //only want to service HTML error pages
            AntPathRequestMatcher.antMatcher("/error")
        )));
}

http.authorizeHttpRequests(cfg->cfg
    .requestMatchers(regexMatcher(HttpMethod.GET, ".+(.css|.jsp|.gif)$"))
    .permitAll()
    .anyRequest().authenticated() ③
);

http.formLogin(cfg->cfg
    .permitAll() //applies permitAll to standard login URIs
    .successForwardUrl("/h2-console") ⑥
);

http.csrf(cfg->cfg.ignoringRequestMatchers(antMatcher("/h2-console/**"))); ④
http.headers(cfg-> cfg.frameOptions(fo->fo.disable())); ⑤

http.authenticationManager(authMgr); //reuse applications authz users ⑦
return http.build();
}
```

① injected `AuthenticationManager` bean exposed by API Configuration

② apply filter rules to H2 UI URIs as well as login/logout form

③ require authenticated users by the application to reach the console

④ turn off CSRF only for the H2 console

- ⑤ turn off display constraints for the H2 console
- ⑥ route successful logins to the H2 console
- ⑦ use pre-configured `AuthenticationManager` for authentication to UI

212.7. Form Login

When we attempt to reach a protected URI within the application with FORM authentication active—the FORM authentication form is displayed.

We should be able to enter the site using any of the username/passwords available to the `AuthenticationManager`. At this point in time, it should be `user1/password1`, `user1/password2`, `user2/password1`, `user3/password2`.



If you enter a bad username/password at the point in time you will receive a JDBC error since we have not yet setup the user database.

212.8. H2 Login

Once we get beyond the application FORM login, we are presented with the H2 database login. The JDBC URL should be set to the value of the `spring.datasource.url` property (`jdbc:h2:mem:users`). The default username is "sa" and has no password. These can be changed with the `spring.datasource.username` and `spring.datasource.password` properties.

212.9. H2 Console

Once successfully logged in, we are presented with a basic but functional SQL interface to the in-memory H2 database that will contain our third source of users—which we need to now setup.

The screenshot shows the H2 Console interface at localhost:8080/h2-console/login.do?jsessionid=df4a0a5a013f0f.... The left sidebar shows the database structure with 'INFORMATION_SCHEMA' and 'Users' tables. The main area displays the 'Users' table with one row: 'H2 1.4.200 (2019-10-14)'. Below the interface is a table of important commands:

	Displays this Help Page
	Shows the Command History
	Ctrl+Enter Executes the current SQL statement
	Shift+Enter Executes the SQL statement defined by the text selection
	Ctrl+Space Auto complete
	Disconnects from the database

Sample SQL Script

212.10. Create DB Schema Script

From the point in time when we added the `spring-boot-starter-jdbc` dependency, we were ready to add database schema—which is the definition of tables, columns, indexes, and constraints of our database. Rather than use a default filename, it is good to keep the schemas separated.

The following file is being placed in the `src/main/resources/database` directory of our source tree. It will be accessible to use within the classpath when we restart the application. The bulk of this implementation comes from the [Spring Security Documentation Appendix](#). I have increased the size of the password column to accept longer Bcrypt encoded password hash values.

Example JDBC UserDetails Database Schema

```
--users-schema.ddl ①
drop table authorities if exists; ②
drop table users if exists;

create table users( ③
    username varchar_ignorecase(50) not null primary key,
    password varchar_ignorecase(100) not null,
    enabled boolean not null);

create table authorities ( ④
    username varchar_ignorecase(50) not null,
    authority varchar_ignorecase(50) not null,
    constraint fkAuthorities_users foreign key(username) references users(username)); ⑤
    create unique index ix_auth_username on authorities (username,authority); ⑥
```

① file places in `src/main/resources/database/users-schema.ddl`

② dropping tables that may exist before creating

③ `users` table primarily hosts `username` and `password`

④ `authorities` table will be used for authorizing accesses after successful identity authentication

- ⑤ `foreign key`' constraint enforces that 'user must exist for any authority
- ⑥ `unique index` constraint enforces all authorities are unique per user and places the foreign key to the users table in an efficient index suitable for querying

The schema file can be referenced through the `spring.database.schema` property by prepending `classpath:` to the front of the path.

Example Database Schema Reference

```
spring.datasource.url=jdbc:h2:mem:users
spring.sql.init.schema-locations=classpath:database/users-schema.ddl
```



Spring Security provides a [schema file](#) with the basics of what is included here. The path is also referenced by `JdbcDaoImpl.DEFAULT_USER_SCHEMA_DDL_LOCATION`

212.11. Schema Creation

The following shows an example of the application log when the schema creation in action.

Example Schema Creation

```
Executing SQL script from class path resource [database/users-schema.ddl]
SQL: drop table authorities if exists
SQL: drop table users if exists
SQL: create table users( username varchar_ignorecase(50) not null primary key,
    password varchar_ignorecase(100) not null, enabled boolean not null)
SQL: create table authorities ( username varchar_ignorecase(50) not null,
    authority varchar_ignorecase(50) not null,
    constraint fkAuthorities_users foreign key(username) references users(username))
SQL: create unique index ix_auth_username on authorities (username,authority)
Executed SQL script from class path resource [database/users-schema.ddl] in 48 ms.
H2 console available at '/h2-console'. Database available at 'jdbc:h2:mem:users'
```

212.12. Create User DB Populate Script

The schema file took care of defining tables, columns, relationships, and constraints. With that defined, we can add population of users. The following user passwords take advantage of knowing we are using the `DelegatingPasswordEncoder` and we made `{noop}plaintext` an option at first.

The JDBC `UserDetailsService` requires that all valid users have at least one authority so I have defined a bogus `known` authority to represent the fact the `username` is known.

Example User DB Populate Script

```
--users-populate.sql
insert into users(username, password, enabled) values('user1','{noop}password',true);
insert into users(username, password, enabled) values('user2','{noop}password',true);
```

```
insert into users(username, password, enabled) values('user3', '{noop}password',true);  
  
insert into authorities(username, authority) values('user1','known');  
insert into authorities(username, authority) values('user2','known');  
insert into authorities(username, authority) values('user3','known');
```

We reference the population script thru a property and can place that in the application.properties file.

Example Database Populate Script Reference

```
spring.datasource.url=jdbc:h2:mem:users  
spring.sql.init.schema-locations=classpath:database/users-schema.ddl  
spring.sql.init.data-locations=classpath:database/users-populate.sql
```

212.13. User DB Population

After the wave of schema commands has completed, the row population will take place filling the tables with our users, credentials, etc.

Example User DB Populate

```
Executing SQL script from class path resource [database/users-populate.sql]  
SQL: insert into users(username, password, enabled)  
values('user1','{noop}password',true)  
SQL: insert into users(username, password, enabled)  
values('user2','{noop}password',true)  
SQL: insert into users(username, password, enabled)  
values('user3','{noop}password',true)  
SQL: insert into authorities(username, authority) values('user1','known')  
SQL: insert into authorities(username, authority) values('user2','known')  
SQL: insert into authorities(username, authority) values('user3','known')  
Executed SQL script from class path resource [database/users-populate.sql] in 7 ms.  
H2 console available at '/h2-console'. Database available at 'jdbc:h2:mem:users'
```

212.14. H2 User Access

With the schema created and users populated, we can view the results using the H2 console.

The screenshot shows the H2 Console interface. On the left, a tree view of the database structure includes 'jdbc:h2:mem:users', 'AUTHORITIES', 'USERS', 'INFORMATION_SCHEMA', and 'Users'. The 'USERS' node is expanded. At the top, there are various toolbar icons and dropdown menus for 'Auto commit', 'Max rows', 'Run', 'Run Selected', 'Auto complete', 'Clear', and 'SQL statement'. The SQL statement 'SELECT * FROM USERS;' is entered in the text field. Below the statement, the results are displayed in a table:

USERNAME	PASSWORD	ENABLED
user1	{noop}password	TRUE
user2	{noop}password	TRUE
user3	{noop}password	TRUE

(3 rows, 3 ms)

An 'Edit' button is located at the bottom of the results panel.

212.15. Authenticate Access using JDBC UserDetailsService

We can now authenticate to access to the API using the credentials in this database.

Example Logins using JDBC UserDetailsService

```
$ curl http://localhost:8080/api/anonymous/hello?name=jim -u user1:password  
hello, jim :caller=user1 ①  
  
$ curl http://localhost:8080/api/anonymous/hello?name=jim -u user1:password1  
hello, jim :caller=user1 ②  
  
$ curl http://localhost:8080/api/anonymous/hello?name=jim -u user1:password2  
hello, jim :caller=user1 ③
```

① authenticating using credentials from JDBC UserDetailsService

② authenticating using credentials from directly configured in-memory UserDetailsService

③ authenticating using credentials from injected in-memory UserDetailsService

However, we still have plaintext passwords in the database. Lets look to clean that up.

212.16. Encrypting Passwords

It would be bad practice to leave the user passwords in plaintext when we have the ability to store cryptographic hash values instead. We can do that through Java and the [BCryptPasswordEncoder](#). The follow example shows using a shell script to obtain the encrypted password value.

Bcrypt Plaintext Passwords

```
$ htpasswd -bnBC 10 user1 password | cut -d\: -f2 ① ②  
$2y$10$UvKwrln7xPp35c5sbj.9kuZ9jY9VYg/VylVTu88ZSCYY/YdcnP/Bq  
  
$ htpasswd -bnBC 10 user2 password | cut -d\: -f2
```

```
$2y$10$tYKBY7act5dN.2d7kumuOsHytIJW8i23Ua2Qogcm6OM638IXMmLS
```

```
$ htpasswd -bnBC 10 user3 password | cut -d\: -f2
$2y$10$AH6uepcNasVx1Ye0hXX20.OX4cI3nXX.LsicoDE5G6bCP34URZZF2
```

① script outputs in format `username:encoded-password`

② `cut` command is breaking the line at the ":" character and returning second field with just the encoded value

212.16.1. Updating Database with Encrypted Values

I have updated the populate SQL script to modify the `{noop}` plaintext passwords with their `{bcrypt}` encrypted replacements.

Update Plaintext Passwords with Encrypted Passwords SQL

```
update users
set password='{bcrypt}$2y$10$UvKwrln7xPp35c5sbj.9kuZ9jY9VVg/VylVTu88ZSCYy/YdcdP/Bq'
where username='user1';
update users
set password='{bcrypt}$2y$10$tYKBY7act5dN.2d7kumuOsHytIJW8i23Ua2Qogcm6OM638IXMmLS'
where username='user2';
update users
set password='{bcrypt}$2y$10$AH6uepcNasVx1Ye0hXX20.OX4cI3nXX.LsicoDE5G6bCP34URZZF2'
where username='user3';
```

Don't Store Plaintext or Decode-able Passwords



The choice of replacing the plaintext INSERTs versus using UPDATE is purely a choice made for incremental demonstration. Passwords should always be stored in their Cryptographic Hash form and never in plaintext in a real environment.

212.16.2. H2 View of Encrypted Passwords

Once we restart and run that portion of the SQL, the plaintext `{noop}` passwords have been replaced by `{bcrypt}` encrypted password values in the H2 console.

The screenshot shows the H2 Console interface with the following details:

- URL:** localhost:8080/h2-console/login.do?jsessionid=aebf4d31e86de779841f015da6325d74
- Toolbar:** Includes icons for back, forward, refresh, and various database operations like Auto commit, Max rows (set to 1000), Run Selected, Auto complete, Clear, and SQL statement input.
- Sidebar:** Shows the database schema with tables: jdbc:h2:mem:users, AUTHORITIES, USERS, and INFORMATION_SCHEMA.
- SQL Statement:** SELECT * FROM USERS
- Result Table:** Displays the USERS table with three rows of data:

SELECT * FROM USERS;		
USERNAME	PASSWORD	ENABLED
user1	{bcrypt}\$2y\$10\$UvKwrln7xPp35c5sbj.9kuZ9jY9VVg/VylVTu88ZSCYy/YdcdP/Bq	TRUE
user2	{bcrypt}\$2y\$10\$tYKBY7act5dN.2d7kumuOsHytIJW8i23Ua2Qogcm6OM638IXMmLS	TRUE
user3	{bcrypt}\$2y\$10\$AH6uepcNasVx1Ye0hXX20.OX4cI3nXX.LsicoDE5G6bCP34URZZF2	TRUE

(3 rows, 4 ms)

Edit button is present at the bottom of the result table.

Figure 95. H2 User Access to Encrypted User Passwords

Chapter 213. Final Examples

213.1. Authenticate to All Three UserDetailsServices

With all `UserDetailsServices` in place, we are able to login as each user using one of the three sources.

Example Logins to All Three UserDetailsServices

```
$ curl http://localhost:8080/api/authn/hello?name=jim -u user1:password -v ②
> Authorization: Basic dXNlcjE6cGFzc3dvcmQ= ①
hello, jim :caller=user1

$ curl http://localhost:8080/api/authn/hello?name=jim -u user2:password1 ③
hello, jim :caller=user2

$ curl http://localhost:8080/api/authn/hello?name=jim -u user3:password2 ④
hello, jim :caller=user3
```

- ① we are still sending a base64 encoding of the plaintext password. The cryptographic hash is created server-side.
- ② `password` is from the H2 database
- ③ `password1` is from the original in-memory user details
- ④ `password2` is from the injected in-memory user details

213.2. Authenticate to All Three Users

With the JDBC `UserDetailsService` in place with encoded passwords, we are able to authenticate against all three users.

Example Logins to Encrypted UserDetails

```
$ curl http://localhost:8080/api/authn/hello?name=jim -u user1:password ①
hello, jim :caller=user1

$ curl http://localhost:8080/api/authn/hello?name=jim -u user2:password ①
hello, jim :caller=user2

$ curl http://localhost:8080/api/authn/hello?name=jim -u user3:password ①
hello, jim :caller=user3
```

- ① three separate user credentials stored in H2 database

Chapter 214. Summary

In this module, we learned:

- the various interfaces and object purpose that are part of the Spring authentication framework
- how to wire up an `AuthenticationManager` with `AuthenticationProviders` to implement authentication for a configured security filter chain
- how to implement `AuthenticationProviders` using only `PasswordEncoder` and `UserDetailsService` primitives
- how to implement in-memory `UserDetailsService`
- how to implement a database-backed `UserDetailsService`
- how to encode and match encrypted password hashes
- how to configure security to display the H2 UI and allow it to be functional

Authorization

copyright © 2024 jim stafford (jim.stafford@jhu.edu)

Chapter 215. Introduction

We have spent a significant amount of time to date making sure we are identifying the caller, how to identify the caller, restricting access based on being properly authenticated, and the management of multiple users. In this lecture we are going to focus on expanding authorization constraints to both roles and permission-based authorities.

215.1. Goals

You will learn:

- the purpose of authorities, roles, and permissions
- how to express authorization constraints using URI-based and annotation-based constraints
- how the enforcement of the constraints is accomplished
- how to potentially customize the enforcement of constraints

215.2. Objectives

At the conclusion of this lecture and related exercises, you will be able to:

1. define the purpose of a role-based and permission-based authority
2. identify how to construct an `AuthorizationManager` and supply criteria to make access decisions
3. implement URI Path-based authorization constraints
4. implement annotation-based authorization constraints
5. implement role inheritance
6. implement an `AccessDeniedException` controller advice to hide necessary stack trace information and provide useful error information to the caller
7. identify the detailed capability of expression-based constraints to be able to handle very intricate situations

Chapter 216. Authorities, Roles, Permissions

An authority is a general term used for a value that is granular enough to determine whether a user will be granted access to a resource. There are different techniques for slicing up authorities to match the security requirements of the application. Spring uses roles and permissions as types of authorities.

A role is a coarse-grain authority assigned to the type of user accessing the system and the prototypical uses that they perform. For example ROLE_ADMIN, ROLE_CLERK, or ROLE_CUSTOMER are relative to the roles in a business application.

A permission is a more fine-grain authority that describes the action being performed versus the role of the user. For example "PRICE_CHECK", "PRICE MODIFY", "HOURS_GET", and "HOURS MODIFY" are relative to the actions in a business application.

No matter which is being represented by the authority value, Spring Security looks to grant or deny access to a user based on their assigned authorities and the rules expressed to protect the resources accessed.

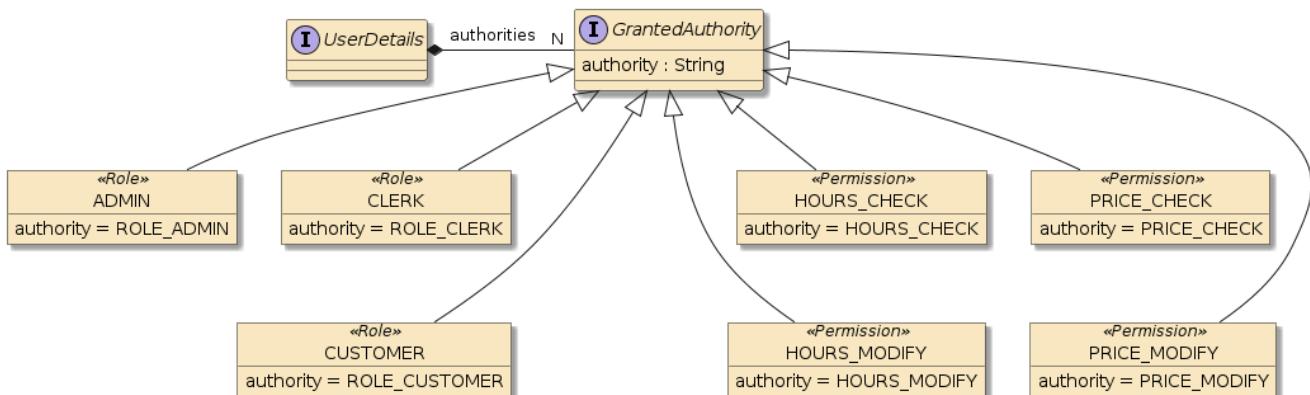


Figure 96. Role and Permission Authorities

Spring represents both roles and permissions using a **GrantedAuthority** class with an authority string carrying the value. Role authority values have, by default, a "ROLE_" prefix, which is a configurable value. Permissions/generic authorities do not have a prefix value. Aside from that, they (roles, permissions, and authorities) look very similar but are not always treated equally.



Spring refers to an authority with **ROLE_** prefix as a "role" and anything with a raw value as an "authority" or "permission". **ROLE_ADMIN** authority represents an **ADMIN** role. **PRICE_CHECK** authority represents a **PRICE_CHECK** permission.

Chapter 217. Authorization Constraint Types

There are two primary ways we can express authorization constraints within Spring Security: path-based and annotation-based.

In the previous sections we have discussed legacy `WebSecurityConfigurer` versus modern Component-based configuration approaches. In this authorization section, we will discuss another dimension of implementation options—legacy `AccessDecisionManager/AccessDecisionVoter` versus modern `AuthorizationManager` approach. Although the legacy `WebSecurityConfigurer` has been removed from Spring Security 6, the legacy `AccessDecisionManager` that was available in Spring Security ≤ 5 is still available but deprecated in favor of the modern `AuthorizationManager` approach. They will both be discussed here since you are likely to encounter the legacy approach in older applications, but the focus will be on the modern approach.

217.1. Path-based Constraints

Path-based constraints are specific to web applications and controller operations since the constraint is expressed against a URI pattern. We define path-based authorizations using the same `HttpSecurity` builder we used with authentication. We can use the legacy `WebSecurityConfigurer` in Spring Security ≤ 5 or the modern Component-based approach in Spring Security ≥ 5 —but never both in the same application.

- legacy `WebSecurityConfigurer` Approach

Authn and Authz HttpSecurity Configuration

```
@Configuration
@Order(0)
@RequiredArgsConstructor
public static class APIConfiguration extends WebSecurityConfigurerAdapter {
    private final UserDetailsService jdbcUserDetailsService;
    @Override
    public void configure(WebSecurity web) throws Exception { ... }
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.requestMatchers(cfg->cfg.antMatchers("/api/**"));
        //...
        http.httpBasic();
        //remaining authn and upcoming authz goes here
    }
}
```

- modern Component-based Approach

Authn and Authz HttpSecurity Configuration

```
@Bean
public WebSecurityCustomizer authzStaticResources() {
    return (web) -> web.ignoring().requestMatchers("/content/**");
}
@Bean
```

```

@Order(0)
public SecurityFilterChain authzSecurityFilters(HttpSecurity http) throws Exception {
    http.securityMatchers(cfg->cfg.requestMatchers("/api/**"));
    //...
    http.httpBasic();
    //remaining authn and upcoming authz goes here
    return http.build();
}

```

The API to instantiate the legacy and modern authorization implementations are roughly the same—especially for annotation-based techniques—however:

- when using deprecated `http.authorizeRequests()`—the deprecated, legacy `AccessDecisionManager` approach will be constructed.
- when using `http.authorizeHttpRequests()`—(Note the extra `Http` in the name)—the modern `AuthorizationManager` will be constructed.

The legacy `WebSecurityConfigurer` examples will be shown with the legacy and deprecated `http.authorizeRequests()` approach. The modern Component-based examples will be shown with the modern `http.authorizeHttpRequests()` approach.

The first example below shows a URI path restricted to the `ADMIN` role. The second example shows a URI path restricted to the `ROLE_ADMIN`, or `ROLE_CLERK`, or `PRICE_CHECK` authorities.



It is worth saying multiple times. Pay attention to the use of the terms "role" and "authority" within Spring Security. `ROLE_X` authority is an "X" role. `X` authority is an "X" permission. The term permission is conceptual. Spring Security only provides builders for roles and authorities.

217.1.1. Legacy AccessDecisionManager Approach

In the legacy `AccessDecisionManager` approach, the (deprecated) `http.authorizeRequests` call is used to define accesses.

Example legacy Path-based Constraints

```

http.authorizeRequests(cfg->cfg.antMatchers(
    "/api/authorities/paths/admin/**")
    .hasRole("ADMIN")); ①
http.authorizeRequests(cfg->cfg.antMatchers(HttpMethod.GET,
    "/api/authorities/paths/price")
    .hasAnyAuthority("PRICE_CHECK", "ROLE_ADMIN", "ROLE_CLERK")); ②

```

① `ROLE_` prefix automatically added to role authorities

② `ROLE_` prefix must be manually added when expressed as a generic authority

217.1.2. Modern AuthorizationManager Approach

In the modern `AuthorizationManager` approach, the `http.authorizeHttpRequests` call is used to define accesses.

The first example below shows a terse use of `hasRole` and `hasAnyAuthority` that will construct a set of `AuthorityAuthorizationManagers` associated with the URI pattern.

Example modern Path-based Constraints

```
http.authorizeHttpRequests(cfg->cfg.requestMatchers(  
    "/api/authorities/paths/admin/**")  
    .hasRole("ADMIN") ①  
);  
http.authorizeHttpRequests(cfg->cfg.requestMatchers(  
    HttpMethod.GET, "/api/authorities/paths/price")  
    .hasAnyAuthority("PRICE_CHECK", "ROLE_ADMIN", "ROLE_CLERK") ②  
);
```

① `ROLE_` prefix automatically added to role authorities. This is building an `AuthorityAuthorizationManager` with an `MvcRequestMatcher` to identify URIs to enforce and an `ADMIN` role to check.

② `ROLE_` prefix must be manually added when expressed as a generic authority. This is building an `AuthorityAuthorizationManager` with an `MvcRequestMatcher` to identify URIs to enforce and a set of authorities ("OR"-d) to check.

The second example below shows a more complex version of `hasAnyAuthority()` to show how `anyOf()` and `allOf()` calls and the aggregate and hierarchical design of the `AuthorityAuthorizationManager` can be used to express more complex access checks.

Example modern Path-based Constraints with Manually Constructed anyOf()

```
http.authorizeHttpRequests(cfg->cfg.requestMatchers(  
    HttpMethod.GET, "/api/authorities/paths/price")  
    .access(AuthorizationManagers.anyOf(  
        AuthorityAuthorizationManager.hasAuthority("PRICE_CHECK"),  
        AuthorityAuthorizationManager.hasRole("ADMIN"),  
        AuthorityAuthorizationManager.hasRole("CLERK")  
    )));
```

Out-of-the-box, path-based authorizations support role inheritance, roles, and permission-based constraints as well as [Spring Expression Language \(SpEL\)](#).

217.2. Annotation-based Constraints

Annotation-based constraints are not directly related to web applications and not associated with URIs. Annotations are placed on the classes and/or methods they are meant to impact. The processing of those annotations has default, built-in behavior that we can augment and modify. The descriptions here are constrained to out-of-the-box capability before trying to adjust anything.

There are three annotation options in Spring:

- @Secured — this was the original, basic annotation Spring used to annotate access controls for classes and/or methods.

```
@Secured("ROLE_ADMIN") ①
@GetMapping(path = "admin", produces = {MediaType.TEXT_PLAIN_VALUE})
public ResponseEntity<String> doAdmin()

@Secured({"ROLE_ADMIN", "ROLE_CLERK", "PRICE_CHECK"}) ②
@GetMapping(path = "price", produces = {MediaType.TEXT_PLAIN_VALUE})
public ResponseEntity<String> checkPrice()
```

① `ROLE_` prefix must be included in string

② Roles and Permissions supported with modern `AuthorizationManager` approach

- JSR 250 — this is an industry standard API for expressing access controls using annotations for classes and/or methods. This is also adopted by JakartaEE.

```
//@RolesAllowed("ROLE_ADMIN") -- Spring Security <= 5
@RolesAllowed("ADMIN") //Spring Security >= 6 ①
@GetMapping(path = "admin", produces = {MediaType.TEXT_PLAIN_VALUE})
public ResponseEntity<String> doAdmin()

//@RolesAllowed({"ROLE_ADMIN", "ROLE_CLERK", "PRICE_CHECK"})
@RolesAllowed({"ADMIN", "CLERK", "PRICE_CHECK"}) ②
@GetMapping(path = "price", produces = {MediaType.TEXT_PLAIN_VALUE})
public ResponseEntity<String> checkPrice()
```

① `ROLE_` prefix no longer included with JSR250 role annotations — automatically added

② Roles and Permissions supported with modern `AuthorizationManager` approach

- Expressions — this annotation capability is based on the powerful Spring Expression Language (SpEL) that allows for ANDing and ORing of multiple values and includes inspection of parameters and current context.

```
@PreAuthorize("hasRole('ADMIN')") ①
@GetMapping(path = "admin", produces = {MediaType.TEXT_PLAIN_VALUE})
public ResponseEntity<String> doAdmin()

...
@PreAuthorize("hasAnyRole('ADMIN','CLERK') or hasAuthority('PRICE_CHECK')") ②
@GetMapping(path = "price", produces = {MediaType.TEXT_PLAIN_VALUE})
public ResponseEntity<String> checkPrice()
```

① `ROLE_` prefix automatically added to role authorities

② `ROLE_` prefix not added to generic authority references

Chapter 218. Setup

The bulk of this lecture will be demonstrating the different techniques for expressing authorization constraints. To do this, I have created four controllers—configured using each technique and an additional `whoAmI` controller to return a string indicating the name of the caller and their authorities.

218.1. Who Am I Controller

To help us demonstrate authorities, I have added a controller to the application that will accept an injected user and return a string that describes who called.

WhoAmI Controller

```
@RestController
@RequestMapping("/api/whoAmI")
public class WhoAmIController {
    @GetMapping(produces={MediaType.TEXT_PLAIN_VALUE})
    public ResponseEntity<String> getCallerInfo(
        @AuthenticationPrincipal UserDetails user) { ①

        List<?> values = (user!=null) ?
            List.of(user.getUsername(), user.getAuthorities()) :
            List.of("null");
        String text = StringUtils.join(values);

        return ResponseEntity.ok(text);
    }
}
```

① `UserDetails` of authenticated caller injected into method call

The controller will return the following when called without credentials.

Anonymous Call

```
$ curl http://localhost:8080/api/whoAmI
[null]
```

The controller will return the following when called with credentials

Authenticated Call

```
$ curl http://localhost:8080/api/whoAmI -u frasier:password
[frasier, [PRICE_CHECK, ROLE_CUSTOMER]]
```

218.2. Demonstration Users

Our user database has been populated with the following users. All have an assigned role (Roles all start with `ROLE_` prefix). One (frasier) has an assigned permission.

```
insert into authorities(username, authority) values('sam','ROLE_ADMIN');
insert into authorities(username, authority) values('rebecca','ROLE_ADMIN');

insert into authorities(username, authority) values('woody','ROLE_CLERK');
insert into authorities(username, authority) values('carla','ROLE_CLERK');

insert into authorities(username, authority) values('norm','ROLE_CUSTOMER');
insert into authorities(username, authority) values('cliff','ROLE_CUSTOMER');
insert into authorities(username, authority) values('frasier','ROLE_CUSTOMER');
insert into authorities(username, authority) values('frasier','PRICE_CHECK');
```

① frasier is assigned a (non-role) permission

218.3. Core Security FilterChain Setup

The following shows the initial/core SecurityFilterChain setup carried over from earlier examples. We will add to this in a moment.

Core SecurityFilterChain Setup

```
//HttpSecurity http
http.authorizeHttpRequests(cfg->cfg.requestMatchers(
    "/api/whoAmI",
    "/api/authorities/paths/anonymous/**")
    .permitAll());

http.httpBasic(cfg->cfg.realmName("AuthzExample"));
http.formLogin(cfg->cfg.disable());
http.headers(cfg->cfg.disable()); //disabling all security headers
http.csrf(cfg->cfg.disable());
http.cors(cfg-> cfg.configurationSource(
    req->new CorsConfiguration().applyPermitDefaultValues()));
http.sessionManagement(cfg->cfg
    .sessionCreationPolicy(SessionCreationPolicy.STATELESS));
```

- The path-based, legacy `AccessDecisionManager` approach is defined through calls to `http.authorizeRequests`

Core SecurityFilterChain legacy AccessDecisionManager Setup

```
http.authorizeRequests(cfg->cfg.antMatchers(
    "/api/whoami",
    "/api/authorities/paths/anonymous/**")
    .permitAll());
```

```
//more ...
```

- The path-based, modern `AuthorizationManager` approach is defined through calls to `http.authorizeHttpRequests`

Core SecurityFilterChain modern AuthorizationManager Setup

```
//HttpSecurity http
http.authorizeHttpRequests(cfg->cfg.requestMatchers(
    "/api/whoAmI",
    "/api/authorities/paths/anonymous/**")
    .permitAll());
//more ...
```

218.4. Controller Operations

The controllers in this overall example will accept API requests and delegate the call to the `WhoAmIController`. Many of the operations look like the snippet example below—but with a different URI.

PathAuthoritiesController Snippet

```
@RestController
@RequestMapping("/api/authorities/paths")
@RequiredArgsConstructor
public class PathAuthoritiesController {
    private final WhoAmIController whoAmI; ①

    @GetMapping(path = "admin", produces = {MediaType.TEXT_PLAIN_VALUE})
    public ResponseEntity<String> doAdmin(
        @AuthenticationPrincipal UserDetails user) {
        return whoAmI.getCallerInfo(user); ②
    }
}
```

① `whoAmI` controller injected into each controller to provide consistent response with username and authorities

② API-invoked controller delegates to `whoAmI` controller along with injected `UserDetails`

Chapter 219. Path-based Authorizations

In this example, I will demonstrate how to apply security constraints on controller methods based on the URI used to invoke them. This is very similar to the security constraints of legacy servlet applications.

The following snippet shows a summary of the URIs in the controller we will be implementing.

Controller URI Summary Snippet

```
@RequestMapping("/api/authorities/paths")
@GetMapping(path = "admin", produces = {MediaType.TEXT_PLAIN_VALUE})
@GetMapping(path = "clerk", produces = {MediaType.TEXT_PLAIN_VALUE})
@GetMapping(path = "customer", produces = {MediaType.TEXT_PLAIN_VALUE})
@GetMapping(path = "price", produces = {MediaType.TEXT_PLAIN_VALUE})
@GetMapping(path = "authn", produces = {MediaType.TEXT_PLAIN_VALUE})
@GetMapping(path = "anonymous", produces = {MediaType.TEXT_PLAIN_VALUE})
@GetMapping(path = "nobody", produces = {MediaType.TEXT_PLAIN_VALUE})
```

219.1. Path-based Role Authorization Constraints

We have the option to apply path-based authorization constraints using roles. The following example locks down three URIs to one or more roles each.

- legacy `AccessDecisionManager` Approach

Example Path Role Authorization Constraints — Legacy AccessDecisionManager Approach

```
http.authorizeRequests(cfg->cfg.antMatchers(
    "/api/authorities/paths/admin/**")
    .hasRole("ADMIN")); ①
http.authorizeRequests(cfg->cfg.antMatchers(
    "/api/authorities/paths/clerk/**")
    .hasAnyRole("ADMIN", "CLERK")); ②
http.authorizeRequests(cfg->cfg.antMatchers(
    "/api/authorities/paths/customer/**")
    .hasAnyRole("CUSTOMER")); ③
```

① `admin` URI may only be called by callers having role `ADMIN` (or `ROLE_ADMIN` authority)

② `clerk` URI may only be called by callers having either the `ADMIN` or `CLERK` roles (or `ROLE_ADMIN` or `ROLE_CLERK` authorities)

③ `customer` URI may only be called by callers having the role `CUSTOMER` (or `ROLE_CUSTOMER` authority)

- modern `AuthorizationManager` approach

```
http.authorizeHttpRequests(cfg->cfg.requestMatchers(  
    "/api/authorities/paths/admin/**")  
    .hasRole("ADMIN")); ①  
http.authorizeHttpRequests(cfg->cfg.requestMatchers(  
    "/api/authorities/paths/clerk/**")  
    .hasAnyRole("ADMIN", "CLERK")); ②  
http.authorizeHttpRequests(cfg->cfg.requestMatchers(  
    "/api/authorities/paths/customer/**")  
    .hasAnyRole("CUSTOMER")); ③
```

① `admin` URI may only be called by callers having role `ADMIN` (or `ROLE_ADMIN` authority)

② `clerk` URI may only be called by callers having either the `ADMIN` or `CLERK` roles (or `ROLE_ADMIN` or `ROLE_CLERK` authorities)

③ `customer` URI may only be called by callers having the role `CUSTOMER` (or `ROLE_CUSTOMER` authority)

219.2. Example Path-based Role Authorization (Sam)

The following is an example set of calls for `sam`, one of our users with role `ADMIN`. Remember that role `ADMIN` is basically the same as saying authority `ROLE_ADMIN`.

```
$ curl http://localhost:8080/api/authorities/paths/admin -u sam:password ①  
[sam, [ROLE_ADMIN]]  
  
$ curl http://localhost:8080/api/authorities/paths/clerk -u sam:password ②  
[sam, [ROLE_ADMIN]]  
  
$ curl http://localhost:8080/api/authorities/paths/customer -u sam:password ③  
{"timestamp":"2020-07-14T15:12:25.927+00:00", "status":403, "error":"Forbidden",  
 "message":"Forbidden", "path":"/api/authorities/paths/customer"}
```

① `sam` has `ROLE_ADMIN` authority, so `admin` URI can be called

② `sam` has `ROLE_ADMIN` authority and `clerk` URI allows both roles `ADMIN` and `CLERK`

③ `sam` lacks role `CUSTOMER` required to call `customer` URI and is rejected with 403/Forbidden error

219.3. Example Path-based Role Authorization (Woody)

The following is an example set of calls for `woody`, one of our users with role `CLERK`.

```
$ curl http://localhost:8080/api/authorities/paths/admin -u woody:password ①  
{"timestamp":"2020-07-14T15:12:46.808+00:00", "status":403, "error":"Forbidden",  
 "message":"Forbidden", "path":"/api/authorities/paths/admin"}
```

```
$ curl http://localhost:8080/api/authorities/paths/clerk -u woody:password ②  
[woody, [ROLE_CLERK]]  
  
$ curl http://localhost:8080/api/authorities/paths/customer -u woody:password ③  
{"timestamp":"2020-07-14T15:13:04.158+00:00", "status":403, "error":"Forbidden",  
"message":"Forbidden", "path":"/api/authorities/paths/customer"}
```

- ① woody lacks role **ADMIN** required to call **admin** URI and is rejected with 403/Forbidden
- ② woody has **ROLE_CLERK** authority, so **clerk** URI can be called
- ③ woody lacks role **CUSTOMER** required to call **customer** URI and is rejected with 403/Forbidden

Chapter 220. Path-based Authority Permission Constraints

The following example shows how we can assign permission authority constraints. It is also an example of being granular with the HTTP method in addition to the URI expressed.

- legacy `AccessDecisionManager` approach

Path-based Authority Authorization Constraints - Legacy AccessDecisionManager Approach

```
http.authorizeRequests(cfg->cfg.antMatchers(HttpMethod.GET, ①
    "/api/authorities/paths/price")
    .hasAnyAuthority("PRICE_CHECK", "ROLE_ADMIN", "ROLE_CLERK")); ②
```

① definition is limited to GET method for URI `price` URI

② must have permission `PRICE_CHECK` or roles `ADMIN` or `CLERK`

- modern `AuthorizationManager` approach

Path-based Authority Authorization Constraints - Modern AuthorizationManager Approach

```
http.authorizeHttpRequests(cfg->cfg.requestMatchers(
    HttpMethod.GET, "/api/authorities/paths/price") ①
    .hasAnyAuthority("PRICE_CHECK", "ROLE_ADMIN", "ROLE_CLERK")); ②
```

① definition is limited to GET method for URI `price` URI

② must have permission `PRICE_CHECK` or roles `ADMIN` or `CLERK`

220.1. Path-based Authority Permission (Norm)

The following example shows one of our users with the `CUSTOMER` role being rejected from calling the `GET price` URI.

Path-based Authority Permission (Norm)

```
$ curl http://localhost:8080/api/authorities/paths/customer -u norm:password ①
[norm, [ROLE_CUSTOMER]]  
  
$ curl http://localhost:8080/api/authorities/paths/price -u norm:password ②
{"timestamp": "2020-07-14T15:13:38.598+00:00", "status": 403, "error": "Forbidden",
 "message": "Forbidden", "path": "/api/authorities/paths/price"}
```

① `norm` has role `CUSTOMER` required to call `customer` URI

② `norm` lacks the `ROLE_ADMIN`, `ROLE_CLERK`, and `PRICE_CHECK` authorities required to invoke the `GET price` URI

220.2. Path-based Authority Permission (Frasier)

The following example shows one of our users with the `CUSTOMER` role and `PRICE_CHECK` permission. This user can call both the `customer` and `GET price` URIs.

Path-based Authority Permission (Frasier)

```
$ curl http://localhost:8080/api/authorities/paths/customer -u frasier:password ①  
[frasier, [PRICE_CHECK, ROLE_CUSTOMER]]  
  
$ curl http://localhost:8080/api/authorities/paths/price -u frasier:password ②  
[frasier, [PRICE_CHECK, ROLE_CUSTOMER]]
```

① `frasier` has the `CUSTOMER` role assigned required to call `customer` URI

② `frasier` has the `PRICE_CHECK` permission assigned required to call `GET price` URI

220.3. Path-based Authority Permission (Sam and Woody)

The following example shows that users with the `ADMIN` and `CLERK` roles are able to call the `GET price` URI.

Path-based Authority Permission (Sam and Woody)

```
$ curl http://localhost:8080/api/authorities/paths/price -u sam:password ①  
[sam, [ROLE_ADMIN]]  
  
$ curl http://localhost:8080/api/authorities/paths/price -u woody:password ②  
[woody, [ROLE_CLERK]]
```

① `sam` is assigned the `ADMIN` role required to call the `GET price` URI

② `woody` is assigned the `CLERK` role required to call the `GET price` URI

220.4. Other Path Constraints

We can add a few other path constraints that do not directly relate to roles. For example, we can exclude or enable a URI for all callers.

- legacy `AccessDecisionManager` approach

Other Path Constraints — Legacy AccessDecisionManager Approach

```
http.authorizeRequests(cfg->cfg.antMatchers(  
    "/api/authorities/paths/nobody/**")  
    .denyAll()); ①  
http.authorizeRequests(cfg->cfg.antMatchers(  
    "/api/authorities/paths/authn/**")
```

```
.authenticated()); ②
```

- ① all callers of the `nobody` URI will be denied
- ② all authenticated callers of the `authn` URI will be accepted

- modern `AuthorizationManager` approach

Other Path Constraints—Modern AuthorizationManager Approach

```
http.authorizeHttpRequests(cfg->cfg.requestMatchers(  
    "/api/authorities/paths/nobody/**")  
    .denyAll());  
http.authorizeHttpRequests(cfg->cfg.requestMatchers(  
    "/api/authorities/paths/authn/**")  
    .authenticated()); //thru customizer.builder ①
```

- ① `authenticated()` call constructs `AuthorizationManager` requiring authenticated caller

A few of the manual forms of configuring path-based access control are shown below to again highlight what the builder methods are conveniently constructing.

Other Path Constraints—Modern AuthorizationManager Approach

```
http.authorizeHttpRequests(cfg->cfg.requestMatchers(  
    "/api/authorities/paths/authn/**")  
    .access(new AuthenticatedAuthorizationManager<>())); //using ctor ①
```

- ① `AuthenticatedAuthorizationManager` default to requiring authentication

Other Path Constraints—Modern AuthorizationManager Approach

```
http.authorizeHttpRequests(cfg->cfg.requestMatchers(  
    "/api/authorities/paths/authn/**")  
    .access(AuthenticatedAuthorizationManager.authenticated())); //thru builder  
①
```

- ① builder constructs default `AuthenticatedAuthorizationManager` instance

220.5. Other Path Constraints Usage

The following example shows a caller attempting to access the URIs that either deny all callers or accept all authenticated callers

```
$ curl http://localhost:8080/api/authorities/paths/authn -u frasier:password ①  
[frasier, [PRICE_CHECK, ROLE_CUSTOMER]]  
  
$ curl http://localhost:8080/api/authorities/paths/nobody -u frasier:password ②  
{"timestamp":"2020-07-14T18:09:38.669+00:00","status":403,  
"error":"Forbidden","message":"Forbidden","path":"/api/authorities/paths/nobody"}
```

```
$ curl http://localhost:8080/api/authorities/paths/authn ③
{"timestamp":"2020-07-14T18:15:24.945+00:00","status":401,
"error":"Unauthorized","message":"Unauthorized","path":"/api/authorities/paths/authn"}
```

- ① **frasier** was able to access the **authn** URI because he was authenticated
- ② **frasier** was not able to access the **nobody** URI because all have been denied for that URI
- ③ anonymous user was not able to access the **authn** URI because they were not authenticated

Chapter 221. Authorization

With that example in place, we can look behind the scenes to see how this occurred.

221.1. Review: FilterSecurityInterceptor At End of Chain

If you remember when we inspected the filter chain setup for our API during the breakpoint in `FilterChainProxy.doFilterInternal()`—there was a `FilterSecurityInterceptor` at the end of the chain. This is where our path-based authorization constraints get carried out.

```
211     FirewalledRequest firewallRequest = this.firewall.getFirewalledRequest((HttpServletRequest) request); firewalledRequest.setPath("path");
212     HttpServletResponse firewallResponse = this.firewall.getFirewalledResponse((HttpServletResponse) response);
213     List<Filter> filters = getFilters(firewallRequest); firewallRequest: "FirewalledRequest[ org.apache.catalina.connector.RequestFacade@596aa251]"
214     if (filters == null || filters.size() == 0) { filters: size = 12
215         if (logger.isTraceEnabled()) {
216             logger.trace(LogMessage.of(() -> "No security for " + requestLine(firewallRequest)));
217         }
218         firewallRequest.reset();
219         this.filterChainDecorator.decorate(chain).doFilter(firewallRequest, firewallResponse);
220         return;
221     }
222     if (logger.isDebugEnabled()) {
223         logger.debug(LogMessage.of(() -> "Securing " + requestLine(firewallRequest)));
224     }
225     FilterChain reset = (req, res) -> {
226 }
```

Evaluate expression (e) or add a watch (o)
> firewallRequest = {StrictHttpFirewall\$StrictFirewalledRequest@9991} "FirewalledRequest[org.apache.catalina.connector.RequestFacade@596aa251]"
> firewallResponse = {FirewalledResponse@12878}
✓ filters = {ArrayList@12879} size = 12
> 0 = {DisableEncodeUrlFilter@12884}
> 1 = {WebAsyncManagerIntegrationFilter@12885}
> 2 = {SecurityContextHolderFilter@12886}
> 3 = {CorsFilter@12887}
> 4 = {LogoutFilter@12888}
> 5 = {BasicAuthenticationFilter@12889}
> 6 = {RequestCacheAwareFilter@12890}
> 7 = {SecurityContextHolderAwareRequestFilter@12891}
> 8 = {AnonymousAuthenticationFilter@12892}
> 9 = {SessionManagementFilter@12893}
> 10 = {ExceptionTranslationFilter@12894}
> 11 = {AuthorizationFilter@12895}

Authz Filter

A red arrow points from the text "Authz Filter" to the `AuthorizationFilter@12895` entry in the variable list.

Figure 97. Review: FilterSecurityInterceptor At End of Chain

221.2. Attempt Authorization Call

We can set a breakpoint in the `AuthorizationFilter.doFilter()` method to observe the authorization process.

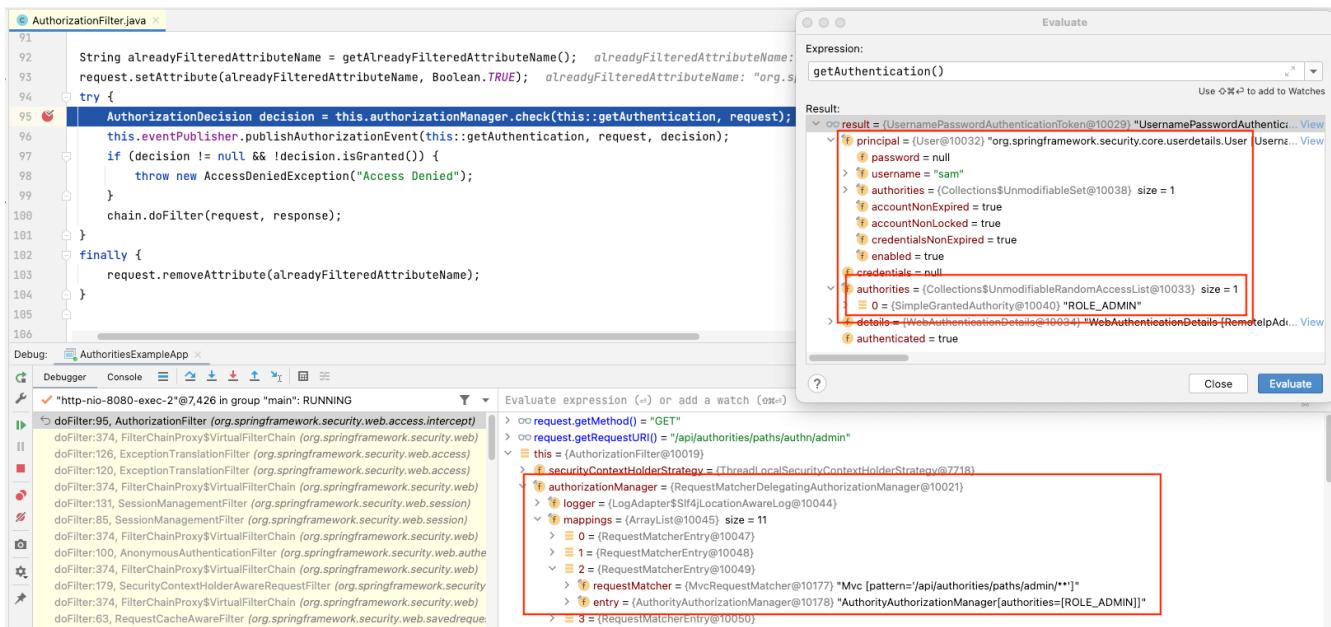


Figure 98. Authorization Call

221.3. FilterSecurityInterceptor Calls

- the `AuthorizationFilter` is at the end of the Security FilterChain and calls the `AuthorizationManager` to decide whether the caller has access to the target object. The call quietly returns without an exception if accepted and throws an `AccessDeniedException` if denied.
- the first `AuthorizationManager` is a `RequestMatcherDelegatingAuthorizationManager` that contains an ordered list of `AuthorizationManagers` that will be matched based upon the `RequestMatcher` expressions supplied.
- the matched `AuthorizationManager` is given a chance to determine user access based upon its definition.
- each matched `AuthorizationManager` returns an answer that is either granted, denied, or abstain.

Depending upon the parent/aggregate `AuthorizationManager`, one or all of the managers have to return a granted access.

Chapter 222. Role Inheritance

Role inheritance provides an alternative to listing individual roles per URI constraint. Let's take our case of `sam` with the `ADMIN` role. He is forbidden from calling the `customer` URI.

Admin Forbidden from Calling customer URI

```
$ curl http://localhost:8080/api/authorities/paths/customer -u sam:password
{"timestamp":"2020-07-14T20:15:19.931+00:00","status":403,"error":"Forbidden",
 "message":"Forbidden","path":"/api/authorities/paths/customer"}
```

222.1. Role Inheritance Definition

We can define a `@Bean` that provides a `RoleHierarchy` expressing which roles inherit from other roles. The syntax to this constructor is a String—based on the legacy XML definition interface.

Example Role Inheritance Definition

```
@Bean
public RoleHierarchy roleHierarchy() {
    return RoleHierarchyImpl.withDefaultRolePrefix()
        .role("ADMIN").implies("CLERK") ①
        .role("CLERK").implies("CUSTOMER") ②
        .build();
}
```

① role `ADMIN` will inherit all access applied to role `CLERK`

② role `CLERK` will inherit all access applied to role `CUSTOMER`

The `RoleHierarchy` component is automatically picked up by the various builders and placed into the runtime `AuthorityAuthorizationManagers`.

Automatically Injecting RoleHierarchy

```
...
http.authorizeHttpRequests(cfg->cfg.requestMatchers(
    "/api/authorities/paths/customer/**")
    .hasAnyRole("CUSTOMER"));
...
```

If manually instantiating an `AuthorityAuthorizationManager`, you can also manually assign the `RoleHierarchy` by injecting it into the `@Bean` factory and adding it to the builder.

Manually Assigning RoleHierarchy

```
@Bean
@Order(0)
public SecurityFilterChain authzSecurityFilters(HttpSecurity http,
```

```

RoleHierarchy roleHierarchy) throws Exception {

    //builder for use with custom access examples
    UnaryOperator<AuthorityAuthorizationManager> withRoleHierarchy = (autho)->{
        autho.setRoleHierarchy(roleHierarchy);
        return autho;
    };

    http.authorizeHttpRequests(cfg->cfg.requestMatchers(
        "/api/authorities/paths/customer/**")
        .access(withRoleHierarchy.apply(
            AuthorityAuthorizationManager.hasAnyRole("CUSTOMER"))));
}

```

With the above `@Bean` in place and restarting our application, users with role `ADMIN` or `CLERK` are able to invoke the `customer` URI.

Admin Inherits CUSTOMER ROLE

```
$ curl http://localhost:8080/api/authorities/paths/customer -u sam:password
[sam, [ROLE_ADMIN]]
```



As you will see in some of the next sections, `RoleHierarchy` is automatically picked up for some of the annotations as well.

Chapter 223. @Secured

As stated before, URIs are one way to identify a target meant for access control. However, it is not always the case that we are protecting a controller or that we want to express security constraints so far from the lower-level component method calls needing protection.

We have at least three options when implementing component method-level access control:

- @Secured
- JSR-250
- expressions

I will cover @Secured and JSR-250 first—since they have a similar, basic constraint capability and save expressions to the end.

223.1. Enabling @Secured Annotations

@Secured annotations are disabled by default.

- For legacy `AccessDecisionManager`, we can enable `@Secured` annotations by supplying a `@EnableGlobalMethodSecurity` annotation with `securedEnabled` set to true.

Enabling @Secured in Legacy AccessDecisionManager

```
import org.springframework.security.config.annotation.method.configuration.EnableGlobalMethodSecurity

@Configuration(proxyBeanMethods = false)
@EnableGlobalMethodSecurity(
    securedEnabled = true // @Secured({"ROLE_MEMBER"})
)
@RequiredArgsConstructor
public class SecurityConfiguration {
```

- For modern `AuthorizationManager`, we can enable `@Secured` by supplying a `@EnableMethodSecurity` annotation with `securedEnabled` set to true.

Enabling @Secured in Legacy AccessDecisionManager

```
import org.springframework.security.config.annotation.method.configuration.EnableMethodSecurity

@Configuration(proxyBeanMethods = false)
@EnableMethodSecurity(
    securedEnabled = true // @Secured({"ROLE_MEMBER"})
)
```

```
@RequiredArgsConstructor  
public class ComponentBasedSecurityConfiguration {
```

223.2. @Secured Annotation

We can add the `@Secured` annotation to the class and method level of the targets we want protected. Values are expressed in authority value. Therefore, since the following example requires the `ADMIN` role, we must express it as `ROLE_ADMIN` authority.

Example @Secured Annotation

```
@RestController  
@RequestMapping("/api/authorities/secured")  
@RequiredArgsConstructor  
public class SecuredAuthoritiesController {  
    private final WhoAmIController whoAmI;  
  
    @Secured("ROLE_ADMIN") ①  
    @GetMapping(path = "admin", produces = {MediaType.TEXT_PLAIN_VALUE})  
    public ResponseEntity<String> doAdmin(  
        @AuthenticationPrincipal UserDetails user) {  
        return whoAmI.getCallerInfo(user);  
    }  
}
```

① caller checked for `ROLE_ADMIN` authority when calling `doAdmin` method

223.3. @Secured Annotation Checks

`@Secured` annotation supports requiring one or more authorities in order to invoke a particular method.

Example @Secure Annotation Checks

```
$ curl http://localhost:8080/api/authorities/secured/admin -u sam:password ①  
[sam, [ROLE_ADMIN]]  
$ curl http://localhost:8080/api/authorities/secured/admin -u woody:password ②  
{"timestamp":"2020-07-14T21:11:00.395+00:00", "status":403,  
 "error":"Forbidden", "trace":"org.springframework.security.access.AccessDeniedException:  
 ...(lots!!!)"}
```

① `sam` has the required `ROLE_ADMIN` authority required to invoke `doAdmin`

② `woody` lacks required `ROLE_ADMIN` authority needed to invoke `doAdmin` and is rejected with an `AccessDeniedException` and a very large stack trace

223.4. @Secured Many Roles

`@Secured` will support many roles ORed together.

Example @Secured with Multiple Roles

```
@Secured({"ROLE_ADMIN", "ROLE_CLERK"})
@GetMapping(path = "price", produces = {MediaType.TEXT_PLAIN_VALUE})
public ResponseEntity<String> checkPrice()
```

A user with either **ADMIN** or **CLERK** role will be given access to **checkPrice()**.

Example @Secured checkPrice()

```
$ curl http://localhost:8080/api/authorities/secured/price -u woody:password
[woody, [ROLE_CLERK]]

$ curl http://localhost:8080/api/authorities/secured/price -u sam:password
[sam, [ROLE_ADMIN]]
```

223.5. @Secured Now Processes Roles and Permissions

@Secured now supports both roles and permissions when using modern **AuthorizationManager**. That was not true with **AccessDecisionManager**.

@Secured Annotation for Roles and Permissions using Modern AuthorizationManager

```
@Secured({"ROLE_ADMIN", "ROLE_CLERK", "PRICE_CHECK"}) ①
@GetMapping(path = "price", produces = {MediaType.TEXT_PLAIN_VALUE})
public ResponseEntity<String> checkPrice()
```

① **PRICE_CHECK** permission will be honored

@Secured Annotation Support Roles for Permissions

```
$ curl http://localhost:8080/api/authorities/secured/price -u frasier:password ①
[frasier, [PRICE_CHECK, ROLE_CUSTOMER]]

curl http://localhost:8080/api/authorities/secured/price -u norm:password ②
{"timestamp": "2023-10-
09T20:51:21.264+00:00", "status": 403, "error": "Forbidden", "trace": "org.springframework.security.access.AccessDeniedException: Access Denied (lots ...)"}
```

① **frasier** can call URI **GET paths/price** because he has permission **PRICE_CHECK** and **@Secured** using **AuthorizationManager** supports permissions

② **norm** cannot call URI **GET secured/price** because he does not have permission **PRICE_CHECK**

223.6. @Secured Role Inheritance

Legacy **AccessDecisionManager** does not (or at least did not) support role inheritance for **@Secured**.

```
$ curl http://localhost:8080/api/authorities/paths/clerk -u sam:password ①
[sam, [ROLE_ADMIN]]
```



```
$ curl http://localhost:8080/api/authorities/secured/clerk -u sam:password ②
{"timestamp":"2023-10-
09T20:55:29.651+00:00","status":403,"error":"Forbidden","trace":"org.springframework.s
ecurity.access.AccessDeniedException: Access Denied (lots!!!)"}  

```

① sam is able to call `paths/clerk` URI because of `ADMIN` role inherits access from `CLERK` role

② sam is unable to call `doClerk()` method because `@Secured` does not honor role inheritance

Modern `AuthorizationManager` does support role inheritance for `@Secured`.

```
$ curl http://localhost:8080/api/authorities/secured/clerk -u sam:password && echo
[sam, [ROLE_ADMIN]]
```

Chapter 224. Controller Advice

When using URI-based constraints, 403/Forbidden checks were done before calling the controller and is handled by a default exception advice that limits the amount of data emitted in the response. When using annotation-based constraints, an `AccessDeniedException` is thrown during the call to the controller and is currently missing an exception advice. That causes a very large stack trace to be returned to the caller (abbreviated here with "...(lots!!!)").

Default AccessDeniedException result

```
$ curl http://localhost:8080/api/authorities/secured/clerk -u sam:password ②
{"timestamp":"2020-07-14T21:48:40.063+00:00","status":403,
"error":"Forbidden","trace":"org.springframework.security.access.AccessDeniedException
...(lots!!!)"}
```

224.1. AccessDeniedException Exception Handler

We can correct that information bleed by adding an `@ExceptionHandler` to address `AccessDeniedException`. In the example below I am building a string with the caller's identity and filling in the standard fields for the returned `MessageDTO` used in the error reporting in my `BaseExceptionAdvice`.

AccessDeniedException Exception Handler

```
...
import org.springframework.security.access.AccessDeniedException;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.RestControllerAdvice;

@RestControllerAdvice
public class ExceptionAdvice extends info.ejava.examples.common.web
.BaseExceptionAdvice { ①
    @ExceptionHandler({AccessDeniedException.class}) ②
    public ResponseEntity<MessageDTO> handle(AccessDeniedException ex) {
        String text=String.format("caller[%s] is forbidden from making this request",
                                  getPrincipal());
        return this.buildResponse(HttpStatus.FORBIDDEN, null, text, (Instant)null);
    }

    protected String getPrincipal() {
        try { ③
            return SecurityContextHolder.getContext().getAuthentication().getName();
        } catch (NullPointerException ex) {
            return "null";
        }
    }
}
```

- ① extending base class with helper methods and core set of exception handlers
- ② adding an exception handler to intelligently handle access denial exceptions
- ③ `SecurityContextHolder` provides `Authentication` object for current caller

224.2. AccessDeniedException Exception Result

With the above `@ExceptionAdvice` in place, the stack trace from the `AccessDeniedException` has been reduced to the following useful information returned to the caller. The caller is told, what they called and who the caller identity was when they called.

AccessDeniedException Filtered through @ExceptionAdvice

```
$ curl http://localhost:8080/api/authorities/secured/clerk -u sam:password
{"url":"http://localhost:8080/api/authorities/secured/clerk","method":"GET","statusCod
e":403,"statusName":"FORBIDDEN","message":"Forbidden","description":"caller[sam] is
forbidden from making this request","timestamp":"2023-10-10T01:18:20.080250Z"}
```

Chapter 225. JSR-250

JSR-250 is an industry Java standard — also adopted by JakartaEE — for expressing common aspects (including authorization constraints) using annotations. It has the ability to express the same things as `@Secured` and a bit more. `@Secured` lacks the ability to express "permit all" and "deny all". We can do that with JSR-250 annotations.

225.1. Enabling JSR-250

JSR-250 authorization annotations are also disabled by default. We can enable them using the `@EnableGlobalMethodSecurity` or `@EnableMethodSecurity` annotations.

- For legacy `AccessDecisionManager`, we can enable them by supplying a `@EnableGlobalMethodSecurity` annotation with `securedEnabled` set to true.

Enabling JSR-250 in Legacy AccessDecisionManager

```
@Configuration(proxyBeanMethods = false)
@EnableGlobalMethodSecurity(
    jsr250Enabled = true // @RolesAllowed({"ROLE_MANAGER"})
)
@RequiredArgsConstructor
public class SecurityConfiguration {
```

- For modern `AuthorizationManager`, we can enable them by supplying a `@EnableMethodSecurity` annotation with `securedEnabled` set to true.

Enabling JSR-250 in Legacy AccessDecisionManager

```
@Configuration(proxyBeanMethods = false)
@EnableMethodSecurity(
    jsr250Enabled = true // @RolesAllowed({"MANAGER"})
)
@RequiredArgsConstructor
public class ComponentBasedSecurityConfiguration {
```

225.2. `@RolesAllowed` Annotation

JSR-250 has a few annotations, but its core `@RolesAllowed` is a 1:1 match for what we can do with `@Secured`. The following example shows the `doAdmin()` method restricted to callers with the admin role, expressed as its `ADMIN` value. Spring 6 change now longer allows the `ROLE_` prefix.

Example `@RolesAllowed` Annotation

```
@RestController
@RequestMapping("/api/authorities/jsr250")
@RequiredArgsConstructor
public class Jsr250AuthoritiesController {
```

```

private final WhoAmIController whoAmI;

@RolesAllowed("ADMIN") ①
@GetMapping(path = "admin", produces = {MediaType.TEXT_PLAIN_VALUE})
public ResponseEntity<String> doAdmin(
    @AuthenticationPrincipal UserDetails user) {
    return whoAmI.getCallerInfo(user);
}

```

① Spring 6 no longer allows `ROLE_` prefix

225.3. `@RolesAllowed` Annotation Checks

The `@RolesAllowed` annotation is restricting callers of `doAdmin()` to have authority `ROLE_ADMIN`.

@RolesAllowed Annotation Checks

```

$ curl http://localhost:8080/api/authorities/jsr250/admin -u sam:password ①
[sam, [ROLE_ADMIN]]

$ curl http://localhost:8080/api/authorities/jsr250/admin -u woody:password ②
{"url": "http://localhost:8080/api/authorities/jsr250/admin", "method": "GET", "statusCode": 403, "statusName": "FORBIDDEN", "message": "Forbidden", "description": "caller[woody] is forbidden from making this request", "timestamp": "2023-10-10T01:24:59.185Z"} 

```

① `sam` can invoke `doAdmin()` because he has the `ROLE_ADMIN` authority

② `woody` cannot invoke `doAdmin()` because he does not have the `ROLE_ADMIN` authority

225.4. Multiple Roles

The `@RolesAllowed` annotation can express multiple authorities the caller may have.

```

@RolesAllowed({"ROLE_ADMIN", "ROLE_CLERK", "PRICE_CHECK"})
@GetMapping(path = "price", produces = {MediaType.TEXT_PLAIN_VALUE})
public ResponseEntity<String> checkPrice()

```

225.5. Multiple Role Check

The following shows were both `sam` and `woody` are able to invoke `checkPrice()` because they have one of the required authorities.

JSR-250 Supports ORing of Required Roles

```

$ curl http://localhost:8080/api/authorities/jsr250/price -u sam:password ①
[sam, [ROLE_ADMIN]]

$ curl http://localhost:8080/api/authorities/jsr250/price -u woody:password ②

```

```
[woody, [ROLE_CLERK]]
```

- ① sam can invoke `checkPrice()` because he has the `ROLE_ADMIN` authority
- ② woody can invoke `checkPrice()` because he has the `ROLE_ADMIN` authority

225.6. JSR-250 Does not Support Non-Role Authorities

JSR-250 authorization annotation processing does not support non-Role authorizations. The following example shows where `frasier` is able to call URI `GET paths/price` but unable to call `checkPrice()` of the JSR-250 controller even though it was annotated with one of his authorities.

JSR-250 Does not Support Non-Role Authorities

```
$ curl http://localhost:8080/api/authorities/paths/price -u frasier:password ①
[frasier, [PRICE_CHECK, ROLE_CUSTOMER]]

$ curl http://localhost:8080/api/authorities/jsr250/price -u frasier:password ②
{"url":"http://localhost:8080/api/authorities/jsr250/price","method":"GET","statusCode":403,"statusName":"FORBIDDEN","message":"Forbidden","description":"caller[frasier] is forbidden from making this request","timestamp":"2023-10-10T01:25:49.310610Z"}
```

- ① `frasier` can invoke URI `GET paths/price` because he has the `PRICE_CHECK` authority and URI-based constraints support non-role authorities
- ② `frasier` cannot invoke JSR-250 constrained `checkPrice()` even though he has `PRICE_CHECK` permission because JSR-250 does not support non-role authorities

A reason for the non-support is that `@RollsAllowed("X")` (e.g., `PRICE_CHECK`) gets translated as role name and translated into "ROLE_X" (`ROLE_PRICE_CHECK`) authority value.

225.7. JSR-250 Role Inheritance

Modern `AuthorizationManager` supports role inheritance. I have not been able to re-test legacy `AccessDecisionManager`.

Modern AuthorizationManager Supports Role Inheritance

```
$ curl http://localhost:8080/api/authorities/jsr250/clerk -u sam:password && echo
[sam, [ROLE_ADMIN]]
```

Chapter 226. Expressions

As demonstrated, `@Secured` and JSR-250-based (`@RolesAllowed`) constraints are functional but very basic. If we need more robust handling of constraints we can use Spring Expression Language and Pre-/Post-Constraints. Expression support is enabled by default for modern `AuthorizationManager` when `@EnableMethodSecurity` is supplied but disabled by default for legacy `AccessDecisionManager` when `@EnableGlobalMethodSecurity` is used.

- For legacy `AccessDecisionManager`, we can enable them by supplying a `@EnableGlobalMethodSecurity` annotation with `prePostEnabled` set to true.

Enabling JSR-250 in Legacy AccessDecisionManager

```
@Configuration(proxyBeanMethods = false)
@EnableGlobalMethodSecurity(
    prePostEnabled = true //@PreAuthorize("hasAuthority('ROLE_ADMIN')"),
    @PreAuthorize("hasRole('ADMIN')")
)
@RequiredArgsConstructor
public class SecurityConfiguration {
```

- For modern `AuthorizationManager`, we can enable them by supplying a `@EnableMethodSecurity` annotation with `prePostEnabled` set to true. However, that is the default for this annotation. Therefore, it is unnecessary to define it as `true`.

Enabling JSR-250 in Legacy AccessDecisionManager

```
@Configuration(proxyBeanMethods = false)
@EnableMethodSecurity(
    prePostEnabled = true //@PreAuthorize("hasAuthority('ROLE_ADMIN')"),
    @PreAuthorize("hasRole('ADMIN')")
)
@RequiredArgsConstructor
public class ComponentBasedSecurityConfiguration {
```

226.1. Expression Role Constraint

Expressions support many callable features, and I am only going to scratch the surface here. The primary annotation is `@PreAuthorize` and whatever the constraint is—it is checked prior to calling the method. There are also features to filter inputs and outputs based on flexible configurations. I will be sticking to the authorization basics and not be demonstrating the other features here. Notice that the contents of the string looks like a function call—and it is. The following example constrains the `doAdmin()` method to users with the role `ADMIN`.

Example Expression Role Constraint

```
@RestController
@RequestMapping("/api/authorities/expressions")
```

```

@RequiredArgsConstructor
public class ExpressionsAuthoritiesController {
    private final WhoAmIController whoAmI;

    @PreAuthorize("hasRole('ADMIN')") ①
    @GetMapping(path = "admin", produces = {MediaType.TEXT_PLAIN_VALUE})
    public ResponseEntity<String> doAdmin(
        @AuthenticationPrincipal UserDetails user) {
        return whoAmI.getCallerInfo(user);
    }
}

```

① `hasRole` automatically adds the `ROLE` prefix

226.2. Expression Role Constraint Checks

Much like `@Secured` and JSR-250, the following shows the caller being checked by expression whether they have the `ADMIN` role. The `ROLE_` prefix is automatically applied.

Example Expression Role Constraint

```

$ curl http://localhost:8080/api/authorities/expressions/admin -u sam:password ①
[sam, [ROLE_ADMIN]]

$ curl http://localhost:8080/api/authorities/expressions/admin -u woody:password ②
{"url":"http://localhost:8080/api/authorities/expressions/admin","method":"GET","statusCode":403,"statusName":"FORBIDDEN","message":"Forbidden","description":"caller[woody] is forbidden from making this request","timestamp":"2023-10-10T01:26:38.853Z"}

```

① `sam` can invoke `doAdmin()` because he has the `ADMIN` role

② `woody` cannot invoke `doAdmin()` because he does not have the `ADMIN` role

226.3. Expressions Support Permissions and Role Inheritance

As noted earlier with URI-based constraints, expressions support non-role authorities and role inheritance for both legacy `AccessDecisionManager` and modern `AuthorizationManager`. The following example checks whether the caller has an authority and chooses to manually supply the `ROLE_` prefix.

Expressions Support non-Role Authorities

```

@PreAuthorize("hasAuthority('ROLE_CLERK')")
@GetMapping(path = "clerk", produces = {MediaType.TEXT_PLAIN_VALUE})
public ResponseEntity<String> doClerk()

```

The following execution demonstrates that a caller with `ADMIN` role will be able to call a method that requires the `CLERK` role because we earlier configured `ADMIN` role to inherit all `CLERK` role accesses.

Example Expression Role Inheritance Checks

```
$ curl http://localhost:8080/api/authorities/expressions/clerk -u sam:password  
[sam, [ROLE_ADMIN]]
```

226.4. Supports Permissions and Boolean Logic

Expressions can get very detailed. The following shows two evaluations being called and their result ORed together. The first evaluation checks whether the caller has certain roles. The second checks whether the caller has a certain permission.

Example Evaluation Logic

```
@PreAuthorize("hasAnyRole('ADMIN', 'CLERK') or hasAuthority('PRICE_CHECK')")  
@GetMapping(path = "price", produces = {MediaType.TEXT_PLAIN_VALUE})  
public ResponseEntity<String> checkPrice()
```

Example Evaluation Logic Checks for Role

```
$ curl http://localhost:8080/api/authorities/expressions/price -u sam:password ①  
[sam, [ROLE_ADMIN]]
```

```
$ curl http://localhost:8080/api/authorities/expressions/price -u woody:password ②  
[woody, [ROLE_CLERK]]
```

① `sam` can call `checkPrice()` because he satisfied the `hasAnyRole()` check by having the `ADMIN` role

② `woody` can call `checkPrice()` because he satisfied the `hasAnyRole()` check by having the `CLERK` role

Example Evaluation Logic Checks for Permission

```
$ curl http://localhost:8080/api/authorities/expressions/price -u frasier:password ①  
[frasier, [PRICE_CHECK, ROLE_CUSTOMER]]
```

① `frasier` can call `checkPrice()` because he satisfied the `hasAuthority()` check by having the `PRICE_CHECK` permission

Example Evaluation Logic Checks

```
$ curl http://localhost:8080/api/authorities/expressions/customer -u norm:password ①  
[norm, [ROLE_CUSTOMER]]
```

```
$ curl http://localhost:8080/api/authorities/expressions/price -u norm:password ②  
{"url": "http://localhost:8080/api/authorities/expressions/price", "method": "GET", "statusCode": 403, "statusName": "FORBIDDEN", "message": "Forbidden", "description": "caller[norm] is forbidden from making this request", "timestamp": "2023-10-10T01:27:16.457797Z"}
```

① `norm` can call `doCustomer()` because he satisfied the `hasRole()` check by having the `CUSTOMER` role

② `norm` cannot call `checkPrice()` because failed both the `hasAnyRole()` and `hasAuthority()` checks by not having any of the looked for authorities.

Chapter 227. Summary

In this module, we learned:

- the purpose of authorities, roles, and permissions
- how to express authorization constraints using URI-based and annotation-based constraints
- how to enforcement of the constraints is accomplished
- how the access control framework centers around a legacy `AccessDecisionManager` /`AccessDecisionVoter` and modern `AuthorizationManager` classes
- how to implement role inheritance for URI and expression-based constraints
- to implement an `AccessDeniedException` controller advice to hide necessary stack trace information and provide useful error information to the caller
- expression-based constraints are limitless in what they can express

Enabling HTTPS

copyright © 2024 jim stafford (jim.stafford@jhu.edu)

Chapter 228. Introduction

In all the examples to date (and likely forward), we have been using the HTTP protocol. This has been very easy option to use, but I likely do not have to tell you that straight HTTP is **NOT secure** for use and especially **NOT appropriate** for use with credentials or any other authenticated information.

Hypertext Transfer Protocol Secure (HTTPS)—with trusted certificates—is the secure way to communicate using APIs in modern environments. We still will want the option of simple HTTP in development and most deployment environments provide an external HTTPS proxy that can take care of secure communications with the external clients. However, it will be good to take a short look at how we can enable HTTPS directly within our Spring Boot application.

228.1. Goals

You will learn:

- the basis of how HTTPS forms trusted, private communications
- the difference between self-signed certificates and those signed by a trusted authority
- how to enable HTTPS/TLS within our Spring Boot application

228.2. Objectives

At the conclusion of this lecture and related exercises, you will be able to:

1. define the purpose of a public certificate and private key
2. generate a self-signed certificate for demonstration use
3. enable HTTPS/TLS within Spring Boot
4. optionally implement an HTTP to HTTPS redirect

Chapter 229. HTTP Access

I have cloned the "noauth-security-example" to form the "https-hello-example" and left most of the insides intact. You may remember the ability to execute the following authenticated command.

Example Successful Authentication

```
$ curl -v -X GET http://localhost:8080/api/authn/hello?name=jim -u user:password ①
> GET /api/authn/hello?name=jim HTTP/1.1
> Host: localhost:8080
> Authorization: Basic dXNlcjpwYXNzd29yZA== ②
>
< HTTP/1.1 200
hello, jim
```

① curl supports credentials with `-u` option

② curl Base64 encodes credentials and adds `Authorization` header

We get rejected when no valid credentials are supplied.

Example Rejection of Anonymous

```
$ curl -X GET http://localhost:8080/api/authn/hello?name=jim
{"timestamp":"2020-07-18T14:43:39.670+00:00","status":401,
 "error":"Unauthorized","message":"Unauthorized","path":"/api/authn/hello"}
```

It works as we remember it, but the issue is that our Base64 encoded (`dXNlcjpwYXNzd29yZA==`), plaintext credentials were issued in the clear.

Base64 Encoding is not Encryption

```
> Authorization: Basic dXNlcjpwYXNzd29yZA== ①
...
$ echo -n dXNlcjpwYXNzd29yZA== | base64 -d && echo ②
user:password
```

① credentials are base64 encoded

② base64 encoded credentials can be easily decoded

We can fix that by enabling HTTPS.

Chapter 230. HTTPS

Hypertext Transfer Protocol Secure (HTTPS) is an extension of HTTP encrypted with Transport Layer Security (TLS) for secure communication between endpoints—offering privacy and integrity (i.e., hidden and unmodified). HTTPS formerly offered encryption with the now deprecated Secure Sockets Layer (SSL). Although the SSL name still sticks around, TLS is only supported today. [\[1\]](#) [\[2\]](#)

230.1. HTTPS/TLS

At the heart of HTTPS/TLS are X.509 certificates and the Public Key Infrastructure (PKI). Public keys are made available to describe the owner (subject), the issuer, and digital signatures that prove the contents have not been modified. If the receiver can verify the certificate and trusts the issuer—the communication can continue. [\[3\]](#)

With HTTPS/TLS, there is one-way and two-way option with one-way being the most common. In one-way TLS—only the server contains a certificate and the client is anonymous at the network level. Communications can continue if the client trusts the certificate presented by the server. In two-way TLS, the client also presents a signed certificate that can identify them to the server and form two-way authentication at the network level. Two-way is very secure but not as common except in closed environments (e.g., server-to-server environments with fixed/controlled communications). We will stick to one-way TLS in this lecture.

230.2. Keystores

A keystore is repository of security certificates - both private and public keys. There are two primary types: Public Key Cryptographic Standards (PKCS12) and Java KeyStore (JKS). PKCS12 is an industry standard and JKS is specific to Java. [\[4\]](#) They both have the ability to store multiple certificates and use an alias to identify them. Both use password protection.

There are typically two uses for keystores: your identity (keystore) and the identity of certificates you trust (truststore). The former is used by servers and must be well protected. The latter is necessary for clients. The truststore can be shared but its contents need to be trusted.

230.3. Tools

There are two primary tools when working with certificates and keystores: keytool and openssl.

[keytool](#) comes with the JDK and can easily generate and manage certificates for Java applications. Keytool originally used the JKS format but since Java 9 switched over to PKCS12 format.

[openssl](#) is a standard, open source tool that is not specific to any environment. It is commonly used to generate and convert to/from all types of certificates/keys.

230.4. Self Signed Certificates

The words "trust" and "verify" were used a lot in the paragraphs above when describing certificates.

When we visit various websites—that locked icon next to the "https" URL indicates the certificate presented by the server was verified and came from a trusted source. Only a server with the associated private key could have generated an inspected value that matched the well-known public key.

Verified Server Certificate



Trusted certificates come from sources that are pre-registered in the browsers and Java JRE truststore and are obtained through purchase.

We can generate self-signed certificates that are not immediately trusted until we either ignore checks or enter them into our local browsers and/or truststore(s).

[1] ["HTTPS"](#), Wikipedia

[2] ["Transport Layer Security"](#), Wikipedia

[3] ["Public key certificate"](#), Wikipedia

[4] ["Spring Boot HTTPS"](#), ZetCode, July 2020

Chapter 231. Enable HTTPS/TLS in Spring Boot

To enable HTTPS/TLS in Spring Boot — we must do the following

1. obtain a digital certificate - we will generate a self-signed certificate without purchase or much fanfare
2. add TLS properties to the application
3. optionally add an HTTP to HTTPS redirect - useful in cases where clients forget to set the protocol to `https://` and use `http://` or use the wrong port number.

231.1. Generate Self-signed Certificate

The following example shows the creation of a self-signed certificate using keytool. Refer to the [keytool reference page](#) for details on the options. The following [Java Keytool page](#) provides examples of several use cases. However, that reference does not include coverage of the now necessary `-ext x509` option for `-genkeypair` and [Subject Alternative Name \(SAN\)](#). The specification of the SAN has to be added to your command.

I kept the values of the certificate extremely basic since there is little chance we will ever use this in a trusted environment. Some key points:

- we request that an RSA certificate be generated
- the certificate should be valid for 10 years
- stored in a keystore ("keystore.p12") using alias ("https-hello")
- tracked by a DN ("CN=localhost,...")
- supplied with a SAN X.509 extension to include a DNS ("localhost") and IP ("127.0.0.1") value that the server can legally respond from to be accepted. SAN is a relatively new requirement and takes the place of validating the hostname using Common Name value ("CN=localhost") of the DN.

Generate Self-signed RSA Certificate

```
$ keytool -genkeypair -keyalg RSA -keysize 2048 -validity 3650 \①
-keystore keystore.p12 -storepass password -alias https-hello \②
-ext "SAN:c=DNS:localhost,IP:127.0.0.1" \③
-dname "CN=localhost,OU=Unknown,O=Unknown,L=Unknown,ST=Unknown,C=Unknown" ④
```

① specifying a valid date 10 years (3650 days) in the future

② assigning the alias `https-hello` to what is generated in the keystore

③ assigning Subject Alternative Names that host can legally answer with

④ assigning a Distinguished Name to uniquely track the certificate

Listing Contents of Keystore

```
$ keytool --list -keystore ./keystore.p12 -storepass password -v
Keystore type: PKCS12
Keystore provider: SUN

Your keystore contains 1 entry

Alias name: https-hello
Creation date: Sep 8, 2024
Entry type: PrivateKeyEntry
Certificate chain length: 1
Certificate[1]:
Owner: CN=localhost, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=Unknown
Issuer: CN=localhost, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=Unknown
Serial number: 6e1836ff31a4e3fb
Valid from: Sun Sep 08 06:32:35 EDT 2024 until: Wed Sep 06 06:32:35 EDT 2034
Certificate fingerprints:
    SHA1: A8:2A:93:56:E4:A0:51:CA:21:32:AF:94:F4:FC:05:10:A7:37:47:CE
    SHA256:
50:22:90:94:64:6E:27:B3:AA:27:53:A9:F4:D5:AD:39:D1:E2:97:D0:33:CF:7B:C1:FB:1E:D0:3E:CE
:FB:24:89
Signature algorithm name: SHA256withRSA
Subject Public Key Algorithm: 2048-bit RSA key
Version: 3

Extensions:

#1: ObjectId: 2.5.29.17 Criticality=true
SubjectAlternativeName [
    DNSName: localhost
    IPAddress: 127.0.0.1
]

#2: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
    KeyIdentifier [
        0000: C5 BD 54 7D 57 BB 6A B4    8D B0 EC B0 D4 98 36 86  ..T.W.j.....6.
        0010: FD 38 81 C3                                .8..
    ]
]
```

 I was alerted to the X.509 feature as a new requirement/solution via the following [Stackoverflow post](#). Prior versions of Spring Boot networking dependencies did not have that extension as a resolution requirement.

231.2. Place Keystore in Reference-able Location

The keytool command output a keystore file called `keystore.p12`. I placed that in the resources area of the application — which will be can be referenced at runtime using a classpath reference.

Place Keystore in Location to be Referenced

```
$ tree src/main/resources/  
src/main/resources/  
|-- application.properties  
`-- keystore.p12
```

Incremental Learning Example Only: Don't use Source Tree for Certs



This example is trying hard to be simple and using a classpath for the keystore to be portable. You should already know how to convert the classpath reference to a file or other reference to keep sensitive information protected and away from the code base. **Do not** store credentials or other sensitive information in the `src` tree in a real application as the `src` tree will be stored in CM.

231.3. Add TLS properties

The following shows a minimal set of properties needed to enable TLS. ^[1]

Example TLS properties

```
server.port=8443①  
server.ssl.enabled=true  
server.ssl.key-store=classpath:keystore.p12②  
server.ssl.key-store-password=password③  
server.ssl.key-alias=https-hello
```

① using an alternate port - optional

② referencing keystore in the classpath — could also use a file reference

③ think twice before placing credentials in a properties file



Do not place credentials in CM system

Do not place real credentials in files checked into CM. Have them resolved from a source provided at runtime.



Note the presence of the legacy "ssl" term in the property name even though "ssl" is deprecated, and we are technically setting up "tls".

[1] ["HTTPS using Self-Signed Certificate in Spring Boot"](#), Baeldung, June 2020

Chapter 232. Untrusted Certificate Error

Once we restart the server, we should be able to connect using HTTPS and port 8443. However, there will be a trust error. The following shows the error from curl.

Untrusted Certificate Error

```
$ curl https://localhost:8443/api/authn/hello?name=jim -u user:password
curl: (60) SSL certificate problem: self signed certificate
More details here: https://curl.haxx.se/docs/sslcerts.html
```

curl failed to verify the legitimacy of the server and therefore could not establish a secure connection to it. To learn more about this situation and how to fix it, please visit the web page mentioned above.

232.1. Option: Supply Trusted Certificates

One option is to construct a file of trusted certificates that includes our self-signed certificate. Curl requires this to be in PEM format. We can build that with the openssl pkcs12 command.

Building Example Trusted Certificate Authority

```
openssl pkcs12 -in ./src/main/resources/keystore.p12 -passin pass:password \
-out ./target/certs.pem \
-clcerts -nokeys -nodes ①
```

① certs only, no private keys, not password protected

PEM File is Text File with Base64 Encoded Certificate Information

```
$ cat target/certs.pem
Bag Attributes
    friendlyName: https-hello
    localKeyID: 54 69 6D 65 20 31 37 32 35 37 39 31 35 35 35 32 31 34
subject=C=Unknown, ST=Unknown, L=Unknown, O=Unknown, OU=Unknown, CN=localhost
issuer=C=Unknown, ST=Unknown, L=Unknown, O=Unknown, OU=Unknown, CN=localhost
-----BEGIN CERTIFICATE-----
MIIDnjCCAoagAwIBAgIIbhg2/zGk4/swDQYJKoZIhvcNAQELBQAwbjEQMA4GA1UE
...
yy1zz1mKV0tzKdtW+PteNVDc
-----END CERTIFICATE-----
```

Adding the generated file as our CA trust source with `--cacert`, we are able to get curl to accept the certificate.

Curl Accepting the Server Certificate After Added to Trusted Sources

```
$ curl -v -X GET https://localhost:8443/api/authn/hello?name=jim -u user:password
```

```
--cacert ./target/certs.pem && echo
...
* CAfile: ./target/trust.pem
...
* Server certificate:
*   subject: C=Unknown; ST=Unknown; L=Unknown; O=Unknown; OU=Unknown; CN=localhost
*   SSL certificate verify ok.
...
< HTTP/1.1 200
hello, jim
```

232.2. Option: Accept Self-signed Certificates

curl and browsers have the ability to accept self-signed certificates either by ignoring their inconsistencies or adding them to their truststore.

The following is an example of curl's `-insecure` option (`-k` abbreviation) that will allow us to communicate with a server presenting a certificate that fails validation.

Enable -insecure Option

```
$ curl -kv -X GET https://localhost:8443/api/authn/hello?name=jim -u user:password
* Connected to localhost (::1) port 8443 (#0)
* ALPN, offering h2
* ALPN, offering http/1.1
* successfully set certificate verify locations:
*   CAfile: /etc/ssl/cert.pem
  CApath: none
* TLSv1.2 (OUT), TLS handshake, Client hello (1):
* TLSv1.2 (IN), TLS handshake, Server hello (2):
* TLSv1.2 (IN), TLS handshake, Certificate (11):
* TLSv1.2 (IN), TLS handshake, Server key exchange (12):
* TLSv1.2 (IN), TLS handshake, Server finished (14):
* TLSv1.2 (OUT), TLS handshake, Client key exchange (16):
* TLSv1.2 (OUT), TLS change cipher, Change cipher spec (1):
* TLSv1.2 (OUT), TLS handshake, Finished (20):
* TLSv1.2 (IN), TLS change cipher, Change cipher spec (1):
* TLSv1.2 (IN), TLS handshake, Finished (20):
* SSL connection using TLSv1.2 / ECDHE-RSA-AES256-GCM-SHA384
* ALPN, server did not agree to a protocol
* Server certificate:
*   subject: C=Unknown; ST=Unknown; L=Unknown; O=Unknown; OU=Unknown; CN=localhost
*   start date: Jul 18 13:46:35 2020 GMT
*   expire date: Jul 16 13:46:35 2030 GMT
*   issuer: C=Unknown; ST=Unknown; L=Unknown; O=Unknown; OU=Unknown; CN=localhost
*   SSL certificate verify result: self signed certificate (18), continuing anyway.
*   Server auth using Basic with user 'user'
> GET /api/authn/hello?name=jim HTTP/1.1
> Host: localhost:8443
> Authorization: Basic dXNlcjpwYXNzd29yZA==
```

```
>  
< HTTP/1.1 200  
hello, jim
```

Chapter 233. Optional Redirect

To handle clients that may address our application using the wrong protocol or port number—we can optionally set up a redirect to go from the common port to the TLS port. The following snippet was taken directly from a [ZetCode](#) article, but I have seen this near exact snippet many times elsewhere.

HTTP:8080 ⇒ HTTPS:8443 Redirect

```
@Bean
public ServletWebServerFactory servletContainer() {
    TomcatServletWebServerFactory tomcat = new TomcatServletWebServerFactory() {
        @Override
        protected void postProcessContext(Context context) {
            SecurityConstraint securityConstraint = new SecurityConstraint();
            securityConstraint.setUserConstraint("CONFIDENTIAL");

            SecurityCollection collection = new SecurityCollection();
            collection.addPattern("/");
            securityConstraint.addCollection(collection);
            context.addConstraint(securityConstraint);
        }
    };
}

tomcat.addAdditionalTomcatConnectors(redirectConnector());
return tomcat;
}

private Connector redirectConnector() {
    Connector connector = new Connector("org.apache.coyote.http11.Http11NioProtocol");
    connector.setScheme("http");
    connector.setPort(8080);
    connector.setSecure(false);
    connector.setRedirectPort(8443);
    return connector;
}
```

233.1. HTTP:8080 ⇒ HTTPS:8443 Redirect Example

With the optional redirect in place, the following shows an example of the client being sent from their original <http://localhost:8080> call to <https://localhost:8443>.

```
$ curl -kv -X GET http://localhost:8080/api/authn/hello?name=jim -u user:password
> GET /api/authn/hello?name=jim HTTP/1.1
> Host: localhost:8080
> Authorization: Basic dXNlcjpwYXNzd29yZA==
>
< HTTP/1.1 302 ①
```

```
< Location: https://localhost:8443/api/authn/hello?name=jim ②
```

① HTTP 302/Redirect Returned

② Location header provides the full URL to invoke — including the protocol

233.2. Follow Redirects

Browsers automatically follow redirects, and we can get curl to automatically follow redirects by adding the `--location` option (or `-L` abbreviated). However, security reasons, this will not present the credentials to the redirected location.

Adding `--location-trusted` will perform the `--location` redirect and present the supplied credentials to the new location — with security risk that we did not know the location ahead of time.

The following command snippet shows curl being requested to connect to an HTTP port , receiving a 302/Redirect, and then completing the original command using the URL provided in the `Location` header of the redirect.

Example curl Follow Redirect

```
$ curl -kv -X GET http://localhost:8080/api/authn/hello?name=jim -u user:password  
--location-trusted ①  
...  
* Server auth using Basic with user 'user'  
> GET /api/authn/hello?name=jim HTTP/1.1  
> Host: localhost:8080  
> Authorization: Basic dXNlcjpwYXNzd29yZA==  
...  
< HTTP/1.1 302  
< Location: https://localhost:8443/api/authn/hello?name=jim  
...  
* Issue another request to this URL: 'https://localhost:8443/api/authn/hello?name=jim'  
...  
* Server certificate:  
* subject: C=Unknown; ST=Unknown; L=Unknown; O=Unknown; OU=Unknown; CN=localhost  
...  
* issuer: C=Unknown; ST=Unknown; L=Unknown; O=Unknown; OU=Unknown; CN=localhost  
* SSL certificate verify result: self signed certificate (18), continuing anyway.  
...  
> GET /api/authn/hello?name=jim HTTP/1.1  
> Host: localhost:8443  
> Authorization: Basic dXNlcjpwYXNzd29yZA==  
...  
< HTTP/1.1 200  
hello, jim
```

① `--location-trusted` redirect option causes curl to follow the 302/Redirect response and present credentials

 It is a security risk to (a) send credentials using an unencrypted HTTP connection and then (b) have the credentials issued to a location issued by the server. This was just an example of how to implement server-side redirects and perform a simple test/demonstration. This was not an example of how to securely implement the client-side.

233.3. Caution About Redirects

One note of caution I will give about redirects is the tendency for IntelliJ to leave orphan processes which seems to get worse with the Tomcat redirect in place. Since our targeted interfaces are for API clients—which should have a documented source of how to communicate with our server—there should be no need for the redirect. The redirect is primarily valuable for interfaces that switch between HTTP and HTTPS. We are either all HTTP or all HTTPS and no need to be eclectic.

Chapter 234. Maven Unit Integration Test

The approach to unit integration testing the application with HTTPS enabled is very similar to non-HTTPS techniques. The changes will be primarily in enhancing the `ClientHttpRequestFactory` that we have been defaulting to a simple HTTP version to date.

Former ClientHttpRequestFactory Bean Factory being Replaced

```
@Bean  
ClientHttpRequestFactory requestFactory() {  
    return new SimpleClientHttpRequestFactory();  
}
```

We will be enhancing the `ClientHttpRequestFactory @Bean` factory with an optional `SSLFactory` to control what gets created. `@Autowired(required=false)` is used to make the injected component optional/nullable.

Enhanced ClientHttpRequestFactory Bean Factory with Optional SSLFactory

```
@Bean @Lazy  
public ClientHttpRequestFactory httpsRequestFactory(  
    @Autowired(required = false) SSLFactory sslFactory) { ... }
```

We will support that method with a conditional `SSLFactory @Bean` factory that will be activated when `it.server.trust-store` property is supplied and non-empty. `@ConditionalOnExpression` is used with a `Spring Expression Language (SpEL)` expression designed to fail if the property is not provided or provided without a value.

Conditional SSLFactory

```
@Bean @Lazy  
@ConditionalOnExpression(  
    "!T(org.springframework.util.StringUtils).isEmpty('${it.server.trust-  
store:}')")  
public SSLFactory sslFactory(ResourceLoader resourceLoader, ServerConfig serverConfig)  
    throws IOException { ... }
```

234.1. JUnit @SpringBootTest

The `@SpringBootTest` will bring in the proper configuration and activate:

- the application's `https` profile to activate HTTPS on the server-side
- the tests `ntest` profile to add any properties needed by the `@SpringBootTest`

```
@SpringBootTest(classes= {ClientTestConfiguration.class}, ①  
    webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)  
@ActiveProfiles({"https", "ntest"}) ②
```

```
@Slf4j  
public class HttpsRestTemplateNTest {  
    @Autowired  
    private RestTemplate authnUser;  
    @Autowired  
    private URI authnUrl;
```

- ① test configuration with test-specific components
- ② activate application `https` profile and testing `ntest` profile

It is important to note that using HTTPS is not an intrusive part of the application or test. If we eliminate the `https` and `ntest` profiles, the tests will complete using straight HTTP.

234.2. application-https.properties [REVIEW]

The following is a repeat from above of what the application has defined for the `https` profile.

application-https.properties

```
server.port=8443①  
server.ssl.enabled=true  
server.ssl.key-store=classpath:keystore.p12②  
server.ssl.key-store-password=password③  
server.ssl.key-alias=https-hello
```

234.3. application-ntest.properties

The following snippet shows the `ntest` profile-specific configuration file used to complete the test definition. It primarily supplies the fact that:

- the protocol scheme will be HTTPS
- trustStore properties need for the client-side of communication

For convenience and consistency, I have defined the `ntest` property values using the `https` property values.

application-ntest.properties

```
it.server.scheme=https  
it.server.trust-store=${server.ssl.key-store} ①  
it.server.trust-store-password=${server.ssl.key-store-password} ①
```

- ① directly referencing properties defined in `application.properties`



The keystore/truststore used in this example is for learning and testing. Do not store operational certs in the source tree. Those files end up in the searchable CM system and the JARs with the certs end up in a Nexus repository.

234.4. ServerConfig

For brevity, I will not repeat the common constructs seen in most API and security test configurations. However, the first quiet change to the `@TestConfiguration` is the definition of the `ServerConfig`. It looks very much the same but know that the `it.server.scheme` is applied to the returned object and is not yet in-place within the `Bean` factory method.

```
@Bean @Lazy  
@ConfigurationProperties("it.server") ①  
public ServerConfig itServerConfig(@LocalServerPort int port) { ②  
    return new ServerConfig().withPort(port);  
}
```

① `@ConfigurationProperties` (scheme=https) are applied to what is returned

② `@LocalServerPort` is assigned at startup



Custom port is being applied before @ConfigurationProperties

`@ConfigurationProperties` are assigned to what is returned from the `@Bean` factory. That means that `it.server.port` property would override `@LocalServerPort` if the property was provided.

234.5. Maven Dependencies

Spring 6 updated the networking in `RestTemplate` to use `httpclient5` and a custom SSL Context library for HTTPS communications. `httpclient5` and its TLS extensions require two new dependencies for our test client to support this extra setup.

Spring 6 RestTemplate Required HTTPS Dependencies

```
<!-- needed to set up TLS for RestTemplate -->  
<dependency>  
    <groupId>org.apache.httpcomponents.client5</groupId>  
    <artifactId>httpclient5</artifactId>  
    <scope>test</scope>  
</dependency>  
<dependency>  
    <groupId>io.github.hakky54</groupId>  
    <artifactId>sslcontext-kickstart-for-apache5</artifactId>  
    <scope>test</scope>  
</dependency>
```

234.6. HTTPS ClientHttpRequestFactory

The HTTPS-based `ClientHttpRequestFactory` is built by following [basic instructions in the docs](#) and tweaking to make SSL a runtime option. There are three main objects created:

1. connectionManager using the optional sslFactory
2. httpClient using the connectionManager
3. requestFactory using the httpClient

The requestFactory is returned.

ClientHttpRequestFactory @Bean Factory with Optional SSL Support

```
import nl.altindag.ssl.util.Apache5SslUtils;
import org.apache.hc.client5.http.classic.HttpClient;
import org.apache.hc.client5.http.impl.classic.HttpClients;
import org.apache.hc.client5.http.impl.io.PoolingHttpClientConnectionManager;
import org.apache.hc.client5.http.impl.io.PoolingHttpClientConnectionManagerBuilder;
/*
https://sslcontext-kickstart.com/client/apache5.html
https://github.com/Hakky54/sslcontext-kickstart#apache-5
*/
@Bean @Lazy
public ClientHttpRequestFactory httpsRequestFactory(
    @Autowired(required = false) SSLFactory sslFactory) { ①
    PoolingHttpClientConnectionManagerBuilder builder =
        PoolingHttpClientConnectionManagerBuilder.create();
    PoolingHttpClientConnectionManager connectionManager =
        Optional.ofNullable(sslFactory)
            .map(sf->builder.setSSLSocketFactory(Apache5SslUtils.toSocketFactory(sf))) ②
            .orElse(builder) ③
            .build();

    HttpClient httpsClient = HttpClients.custom()
        .setConnectionManager(connectionManager)
        .build();
    return new HttpComponentsClientHttpRequestFactory(httpsClient);
}
```

① `@Autowired.required=false` allows `sslFactory` to be null if not using SSL

② `Optional` processing path if `sslFactory` is present

③ `Optional` processing path if `sslFactory` is not present

234.7. SSL Factory

The `SSLFactory @Bean` factory is conditional based on the presence of a non-empty `it.server.trust-store` property. If that property does not exist or has an empty value — this `@Bean` factory will not be invoked and the `ClientHttpRequestFactory @Bean` factory will be invoked with a null for the `sslFactory`.

The trustStore is loaded with the help of the Spring `ResourceLoader`. By using this component, we can use `classpath:, file:,` and other resource location syntax in order to construct an `InputStream` to ingest the trustStore.

```
import nl.altindag.ssl.SSLFactory;
import org.springframework.core.io.ResourceLoader;

@Bean @Lazy
@ConditionalOnExpression(
    "!T(org.springframework.util.StringUtils).isEmpty('${it.server.trust-
store}')") ①
public SSLFactory sslFactory(ResourceLoader resourceLoader, ServerConfig serverConfig)
    throws IOException {
    try (InputStream trustStoreStream = resourceLoader
        .getResource(serverConfig.getTrustStore()).getInputStream()) {
        return SSLFactory.builder()
            .withProtocols("TLSv1.2")
            .withTrustMaterial(trustStoreStream, serverConfig.getTrustStorePassword())
            .build();
    } catch (FileNotFoundException ex) {
        throw new IllegalStateException("unable to locate truststore: " +
serverConfig.getTrustStore(), ex);
    }
}
```

① conditional activates factory bean when `it.server.trust-store` is not empty

234.8. JUnit @Test

The core parts of the JUnit test are pretty basic once we have the HTTPS/Authn-enabled `RestTemplate` and `baseUrl` injected.

Review: Standard RestTemplate Construction Used to Date

```
@Bean @Lazy
public RestTemplate authnUser(RestTemplateBuilder builder,
                           ClientHttpRequestFactory requestFactory) {
    RestTemplate restTemplate = builder.requestFactory(
        //used to read streams twice -- enable use of logging filter below
        ()->new BufferingClientHttpRequestFactory(requestFactory))
        .interceptors(new BasicAuthenticationInterceptor(username, password),
                     new RestTemplateLoggingFilter())
        .build();
    return restTemplate;
}
```

From here it is just a normal test, but activity is remote on the server side.

Review: Example Test

```
public class HttpsRestTemplateNTest {
    @Autowired ①
```

```

private RestTemplate authnUser;
@Autowired ②
private URI authnUrl;

@Test
public void user_can_call_authenticated() {
    //given a URL to an endpoint that accepts only authenticated calls
    URI url = UriComponentsBuilder.fromUri(authnUrl)
        .queryParam("name", "jim").build().toUri();

    //when called with an authenticated identity
    ResponseEntity<String> response = authnUser.getForEntity(url, String.class);

    //then expected results returned
    then(response.getStatusCode()).isEqualTo(HttpStatus.OK);
    then(response.getBody()).isEqualTo("hello, jim");
}
}

```

① `RestTemplate` with authentication and HTTPS aspects addressed using filters

② `authnUrl` built from `ServerConfig` and injected into test

Example HTTPS Test Execution

```

INFO ClientTestConfiguration#authnUrl:52 baseUrl=https://localhost:60364
INFO HttpsRestTemplateNTest#setUp:30 baseUrl=https://localhost:60364/api/authn/hello
DEBUG RestTemplate#debug:127 HTTP GET https://localhost:60364/api/authn/hello?name=jim

```

With the `https` and `ntest` profiles disabled, the test reverts to HTTP.

Example HTTPS Test Execution

```

INFO ClientTestConfiguration#authnUrl:52 baseUrl=http://localhost:60587
INFO HttpsRestTemplateNTest#setUp:29 baseUrl=http://localhost:60587/api/authn/hello
DEBUG RestTemplate#debug:127 HTTP GET http://localhost:60587/api/authn/hello?name=jim

```

Chapter 235. Summary

In this module, we learned:

- the basis of how HTTPS forms trusted, private communications
- how to generate a self-signed certificate for demonstration use
- how to enable HTTPS/TLS within our Spring Boot application
- how to add an optional redirect and why it may not be necessary

AutoRentals Assignment 3: Security

copyright © 2024 jim stafford (jim.stafford@jhu.edu)

This is a single assignment that has been broken into incremental, compatible portions based on the completed API assignment. It should be turned in as a single tree of Maven modules that can build from the root level down.

Chapter 236. Assignment Starter

There is a project within `autorentals-starter/assignment3-autorentals-security/autorentals-security-svc` that contains some ground work for the security portions of the assignment. It contains:

1. a set of `@Configuration` classes nested within the `SecurityConfiguration` class. Each `@Configuration` class or construct is profile-constrained to match a section of the assignment.
2. the shell of a secure AutoRentalsService wrapper



It is your choice whether to use the "layered/wrapped" approach (where you implement separate/layered API and security modules) or "embedded/enhanced" approach (where you simply enhance the API solution with the security requirements).

3. a `@Configuration` class that instantiates the correct AutoRentalService under the given profile/runtime context.
4. base unit integration tests that align with the sections of the assignment and pull in base tests from the support module. Each test case activates one or more profiles identified by the assignment.

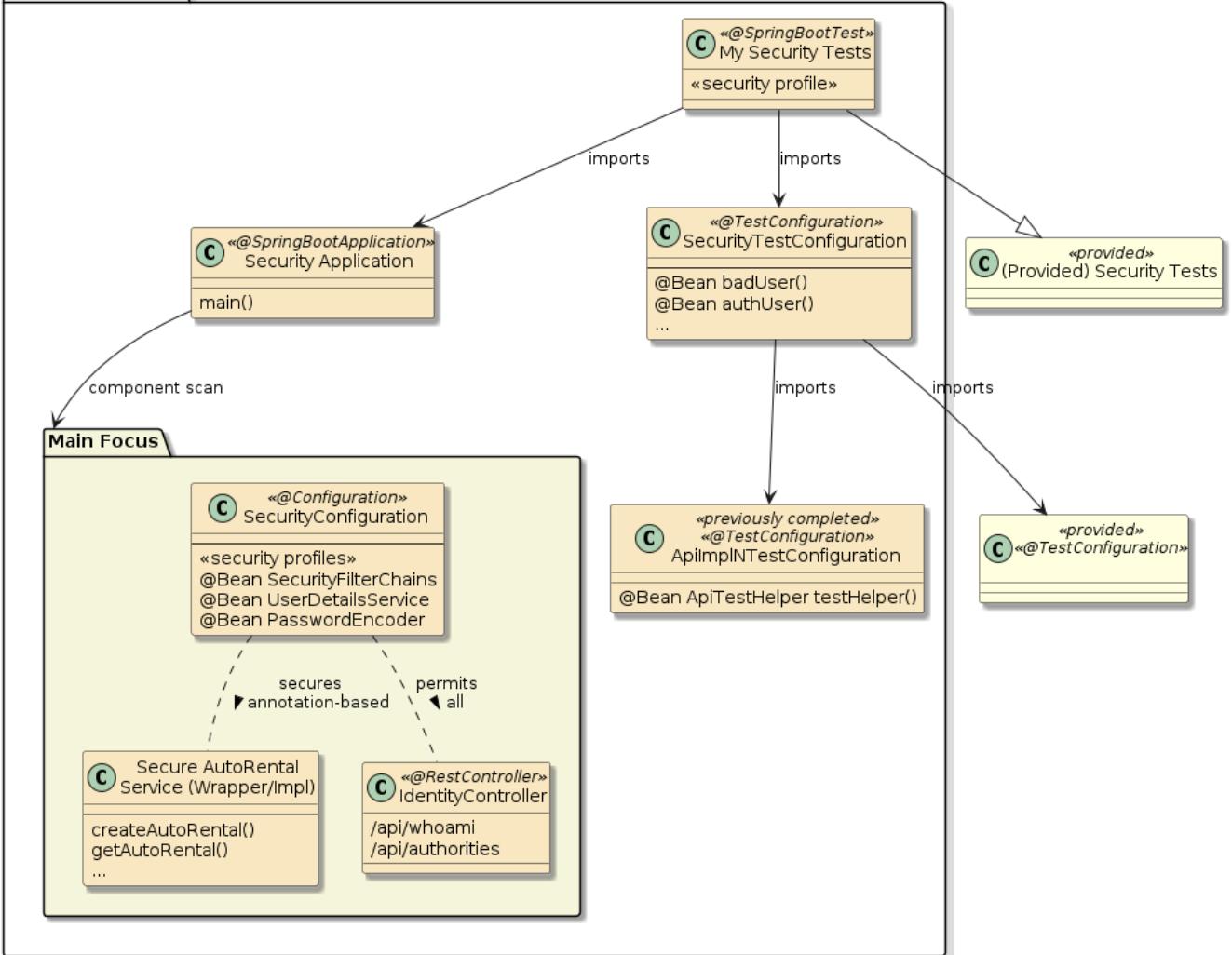


Figure 99. Security Assignment Starter

The meat of the assignment is focused in the following areas:

1. configuring web security beans to meet the different security levels of the assignment. You will use the **component-based** approach. Each profile will start over. You will end up copy/pasting filtering rules forward to follow-on configurations of advancing profiles.
2. the unit tests are well populated
 - a. each of the tests have been **@Disabled** and rely on your test helper from the API tests to map `RentalDTO` references to your `AutoRental DTO` class.
3. For the other server-side portions of the assignment:
 - a. there is a skeletal `IdentityController` that outlines portions of the `whoAmI` and `authorities` endpoints.
 - b. there is a skeletal `SecureAutoRentalsServiceWrapper` and `@Configuration` class that can be used to optionally wrap your assignment 2 implementation. The satisfactory alternative is to populate your existing assignment 2 implementations to meet the assignment 3 requirements. Neither path (layer versus enhance in-place) will get you more or less credit. Choose the path that makes the most sense to you.



The layered approach provides an excuse to separate the solution into

layers of components and provide practice doing so.



The enhance-in-place approach is a more straight forward and realistic development because everything is in one place. It would be rare to split the implementation of a single component across a series of modules/JARs.



If you layer your assignment3 over your assignment2 solution as separate modules, you will need to make sure that the dependencies on assignment2 are vanilla JARs and not Spring Boot executable JARs. Luckily the `ejava-build-parent` made that easy to do with classifying the executable JAR with `bootexec`.

Chapter 237. Assignment Support

The assignment support module(s) in `autorentals-support/autorentals-support-security` again provide some examples and ground work for you to complete the assignment—by adding a dependency. I used a layered approach to secure Autos and Renters. This better highlighted what was needed for security because it removed most of the noise from the assignment 2 functional threads. It also demonstrated some weaving of components within the auto-configuration. Adding the dependency on the `autorenters-support-security-svc` adds this layer to the `autorenters-support-api-svc` components.

The following dependency can replace your current dependency on the `autorenters-support-api-svc`.

autorenters-support-security-svc Dependency

```
<dependency>
    <groupId>info.ejava.assignments.security.autorentals</groupId>
    <artifactId>autorenters-support-security-svc</artifactId>
    <version>${ejava.version}</version>
</dependency>
```

The support module comes with an extensive amount of tests that permit you to focus your attention on the security configuration and security implementation of the AutoRental service. The following test dependency can provide you with many test constructs.

autorenters-support-security-svc Dependency

```
<dependency>
    <groupId>info.ejava.assignments.security.autorentals</groupId>
    <artifactId>autorenters-support-security-svc</artifactId>
    <classifier>tests</classifier>
    <version>${ejava.version}</version>
    <scope>test</scope>
</dependency>
```

237.1. High Level View

Between the support (primary and test) modules and starter examples, most of your focus can be placed on completing the security configuration and service implementation to satisfy the security requirements.

The support module provides

- Auto and Renter services that will operate within your application and will be secured by your security configuration. Necessary internal security checks are included within the Auto and Renter services, but your security configuration will use path-based security access to provide interface access control to these externally provided resources.

The support test modules provide

- `@TestConfiguration` classes that supply the necessary beans for the tests to be completed.
- Test cases that are written to be base classes of `@SpringBootTest` test cases supplied in your assignment. The starter provides most of what you will need for your security tests.

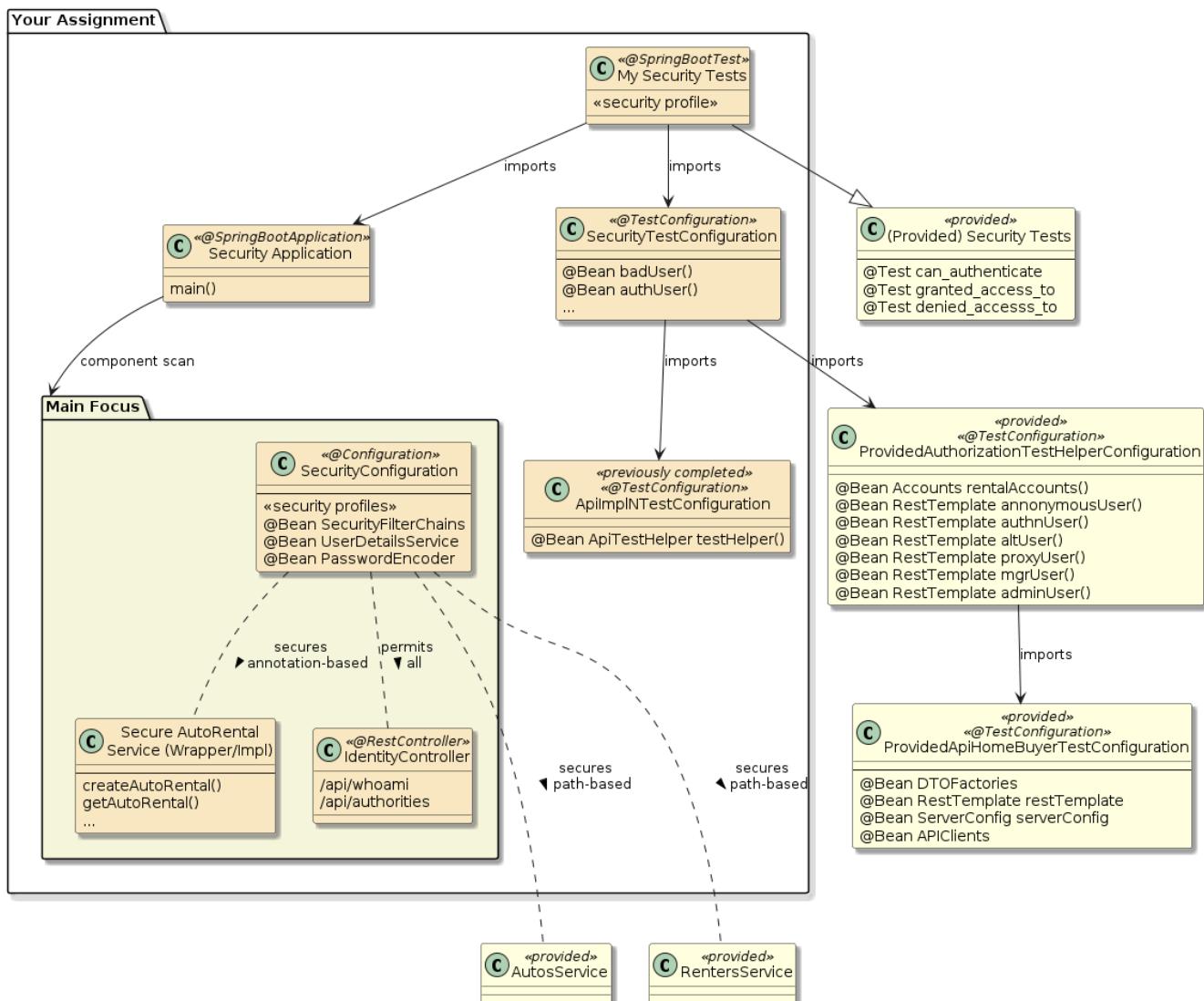


Figure 100. Security Assignment High Level View

Your main focus should be within the security configuration and AutoRentals service implementation classes and running the series of provided tests.

The individual sections of this assignment are associated with one or more Spring profiles. The profiles are activated by your `@SpringBootTest` test cases. The profiles will activate certain test configurations and security configurations your are constructing.

- Run a test
- Fix a test result with either a security configuration or service change
- rinse and repeat



The tests are written to execute from the sub-class in your area. With adhoc navigation, sometimes the IDE can get lost—lose the context of the sub-class and

provide errors as if there were only the base class. If that occurs—make a more direct IDE command to run the sub-class to clear the issue.

Chapter 238. Assignment 3a: Security Authentication

238.1. Anonymous Access

238.1.1. Purpose

In this portion of the assignment, you will demonstrate your knowledge of configuring Spring Web Security for authentication requirements. You will:

1. activate Spring Security
2. create multiple, custom authentication filter chains
3. enable open access to static resources
4. enable anonymous access to certain URIs
5. enforce authenticated access to certain URIs

238.1.2. Overview

In this portion of the assignment you will be activating and configuring the security configuration to require authentication to certain resource operations while enabling access to other resources operations.

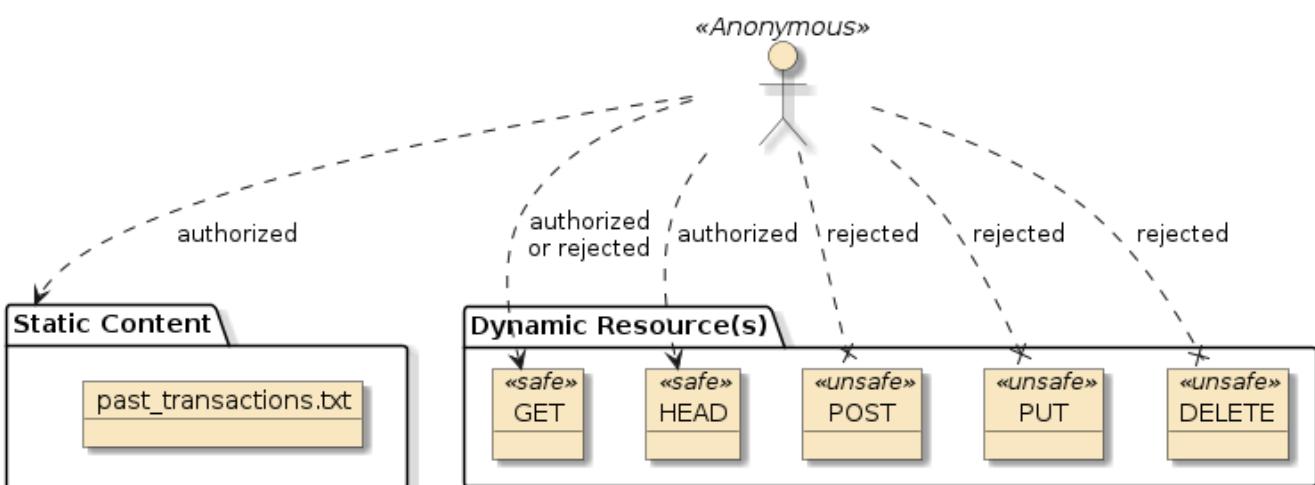


Figure 101. Anonymous Access

URIs

- /api/renters
- /api/renters/{id}
- /api/autos
- /api/autos/{id}
- /api/{your resource}
- /api/{your resource}/id
- /content/*

238.1.3. Requirements

1. Turn off **CSRF** protections.



Configure `HttpSecurity` to disable CSRF processing. This will prevent ambiguity between a CSRF or authorization rejection for non-safe HTTP methods.

2. Add a static text file `past_transactions.txt` that will be made available below the `/content/` URI. Place the following title in the first line so that the following will be made available from a web client.

`past_transactions.txt`

```
$ curl -X GET http://localhost:8080/content/past_transactions.txt
Past AutoRental
```



Static Resources Served from classpath

By default, static content is served out of the classpath from several named locations including `classpath:/static/`, `classpath:/public/`, `classpath:/resources/`, and `classpath:/META-INF/resources/`

3. Configure anonymous access for the following resources methods

- a. static content below `/content`



`/content/**` glob indicates "anything below" `/content`.

- b. **HEAD** for all resources



Configure `HttpSecurity` to permit all **HEAD** calls matching any URI — whether it exists or not.

- c. **GET** for auto and autoRental resources. Leverage the `requestMatcher()` to express the method and `pattern`.



Configure `HttpSecurity` to permit all **GET** calls matching URIs below `autos` and `autoRentals`. Leverage the `requestMatcher()` to express the method and `pattern`.

- d. **POST** for `/api/autos/query` access

4. Configure authenticated access for the following resource operations

- a. **GET** calls for renter resources. No one can gain access to a Renter without being authenticated.

- b. non-safe calls (**POST**, **PUT**, and **DELETE**) for autos, renters, and autoRentals resources; except **POST /api/autos/query**.



Configure `HttpSecurity` to authenticate any request that was not yet explicitly

permitted

5. Create a unit integration test case that verifies (provided)

- a. anonymous user access **granted** to static content
- b. anonymous user access **granted** to a **HEAD** call to auto and renter resources
- c. anonymous user access **granted** to a **GET** call to auto and autoRental resources
- d. anonymous user access **denial** to a **GET** call to renter resources
- e. anonymous user access **denial** to non-safe call to each resource type



Denial must be because of authentication requirements and not because of a CSRF failure. Disable CSRF for all API security configurations.



All tests will use an anonymous caller in this portion of the assignment. Authenticated access is the focus of a later portion of the assignment.

6. Restrict this security configuration to the **anonymous-access** profile and activate that profile while testing this section.

```
@Configuration(proxyBeanMethods = false)
@Profile("anonymous-access") ①
public class PartA1_AnonymousAccess {
```

① restrict configuration changes to the **anonymous-access** profile



```
@SpringBootTest(classes= {...},
@ActiveProfiles({"test", "anonymous-access"}) ①
public class MyA1_AnonymousAccessNTest extends A1_AnonymousAccessNTest
{
```

① activate the **anonymous-access** profile (with any other designed profiles) when executing tests

238.1.4. Grading

Your solution will be evaluated on:

1. activate Spring Security
 - a. whether Spring security has been enabled
2. create multiple, custom authentication filter chains
 - a. whether access is granted or denied for different resource URIs and methods
3. enable open access to static resources
 - a. whether anonymous access is granted to static resources below **/content**

4. enable anonymous access to certain URIs
 - a. whether anonymous access has been granted to dynamic resources for safe (`GET`) calls
5. enforce authenticated access to certain URIs
 - a. whether anonymous access is denied for dynamic resources for unsafe (`POST`, `PUT`, and `DELETE`) calls

238.1.5. Additional Details

1. No accounts are necessary for this portion of the assignment. All testing is performed using an anonymous caller.
2. Turning on TRACE for Spring Web Security may provide helpful narrative to actions taken

```
logging.level.org.springframework.security.web=TRACE
```

3. Setting a breakpoints at or near:
 - `FilterChainProxy.doFilterInternal():214` will allow you to inspect the filterChains put in place by your definitions.
 - `FilterChainProxy.VirtualFilterChain.doFilter():374` will allow you to inspect the work performed by each filter
 - `AuthorizationFilter.doFilter():95,`
`RequestMatcherDelegatingAuthorizationManager.check():74` will allow you to inspect authorization

238.2. Authenticated Access

238.2.1. Purpose

In this portion of the assignment, you will demonstrate your knowledge of authenticating a client and identifying the identity of a caller. You will:

1. add an authenticated identity to `RestTemplate`, or `RestClient` client
2. locate the current authenticated user identity

238.2.2. Overview

In this portion of the assignment you will be authenticating with the API (using `RestTemplate`, or `RestClient`) and tracking the caller's identity within the application.

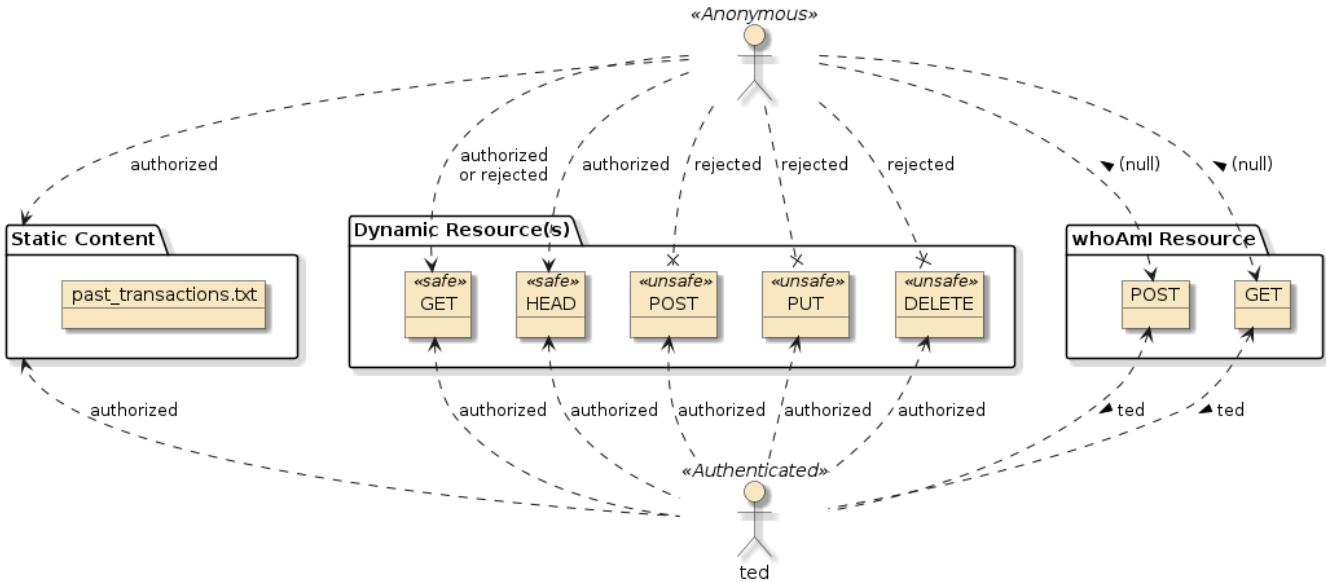


Figure 102. Authenticated Access

You're starting point should be an application with functional Autos, Renters, and AutoRentals API services. Autos and Renters were provided to you. You implemented AutoRentals in assignment2. All functional tests for AutoRentals are passing.

Your focus area is again in the `SecurityConfiguration @Configuration` class. This time, it is within the `@Configuration` that is active during the `authenticated-access` profile. The changes you made for the previous profile will not longer be active. Copy/paste anything that is again needed for these requirements.

Focus Area for this Portion of the Assignment

```
@Configuration(proxyBeanMethods = false)
@Profile({"authenticated-access", "userdetails"})
public class PartA2_AuthenticatedAccess {
```

238.2.3. Requirements

- Configure the application to use a test username/password of `ted/secret` during testing with the `authenticated-access` profile.

Account properties should only be active when using the `authenticated-access` profile.

```
spring.security.user.name: ...
spring.security.user.password: ...
```



Active profiles can be named for individual Test Cases.

```
@SpringBootTest(...)
@ActiveProfiles({"test", "authenticated-access"})
```

Property values can be injected into the test configurations in order to supply known values.

```
@Value("${spring.security.user.name}") ①  
private String username;  
@Value("${spring.security.user.password}")  
private String password;
```

① value injected into property if defined — blank if not defined

2. Configure the application to support **BASIC** authentication.



Configure HttpSecurity to enable HTTP BASIC authentication.

3. Turn off **CSRF** protections.



Configure HttpSecurity to disable CSRF processing.

4. Turn off sessions, and any other filters that prevent API interaction beyond authentication.



Configure HttpSecurity to use Stateless session management and other properties as required.

5. *Add a new resource `api/whoAmI` (provided)*

- a. supply two access methods: **GET** and **POST**. Configure security such that neither require authentication.



Configure HttpSecurity to permit all method requests for `/api/whoAmI`. No matter which HttpMethod is used. Pay attention to the order of the authorize requests rules definition.

- b. both methods must determine the identity of the current caller and return that value to the caller. When called with no authenticated identity, the methods should return a String value of “(null)” (open_paren + the_word_null + close_paren)



You may inject or programmatically lookup the user details for the caller identity within the **server-side** controller method.

6. *Create a set of unit integration tests that demonstrate the following (provided):*

- a. authentication denial when using a known username but bad password



Any attempt to authenticate with a bad credential will result in a **401/UNAUTHORIZED** error no matter if the resource call requires authentication or not.



Credentials can be applied to **RestTemplate** using interceptors.

- b. successful authentication using a valid username/password
 - c. successful identification of the authenticated caller identity using the `whoAmI` resource operations
 - d. successful authenticated access to POST/create auto, renter, and autoRental resource operations
7. Restrict this security configuration to the `authenticated-access` profile and activate that profile during testing this section.



```
@Configuration(proxyBeanMethods = false)
@Profile({"authenticated-access", "userdetails"}) ①
public class PartA2_AuthenticatedAccess {
===
@SpringBootTest(classes={...},
    webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
@ActiveProfiles({"test", "authenticated-access"}) ②
public class MyA2_AuthenticatedAccessNTest extends
A2_AuthenticatedAccessNTest {
```

- ① activated with `authenticated-access` (or upcoming `userdetails`) profile
- ② activating desired profiles

8. Establish a security configuration that is active for the `nosecurity` profile which allows for but does not require authentication to call each resource/method. This will be helpful to simplify some demonstration scenarios where security is not essential.



```
@Configuration(proxyBeanMethods = false)
@Profile("nosecurity") ①
public class PartA2b_NoSecurity {
===
@SpringBootTest(classes= { ... }
@ActiveProfiles({"test", "nosecurity"}) ①
public class MyA2b_NoSecurityNTest extends A2b_NoSecurityNTest {
```

The `security` profile will allow unauthenticated access to operations that are also free of CSRF checks.



```
curl -X POST http://localhost:8080/api/autos -H 'Content-Type: application/json' -d '{ ... }'
{ ... }
```

238.2.4. Grading

Your solution will be evaluated on:

1. add an authenticated identity to `RestTemplate` or `RestClient` client
 - a. whether you have implemented stateless API authentication (`BASIC`) in the server
 - b. whether you have successfully completed authentication using a Java client
 - c. whether you have correctly demonstrated and tested for authentication denial
 - d. whether you have demonstrated granted access to an unsafe methods for the auto, renter, and autoRental resources.
2. locate the current authenticated user identity
 - a. whether your server-side components are able to locate the identity of the current caller (authenticated or not).

238.2.5. Additional Details

238.3. User Details

238.3.1. Purpose

In this portion of the assignment, you will demonstrate your knowledge of assembling a `UserDetailsService` to track the credentials of all users is the service. You will:

1. build a `UserDetailsService` implementation to host user accounts and be used as a source for authenticating users
2. build an injectable `UserDetailsService`
3. encode passwords

238.3.2. Overview

In this portion of the assignment, you will be starting with a security configuration with the authentication requirements of the previous section. The main difference here is that there will be multiple users and your server-side code needs to manage multiple accounts and permit them each to authenticate.

To accomplish this, you will be constructing an `AuthenticationManager` to provide the user credential authentication required by the policies in the `SecurityFilterChain`. Your supplied `UserDetailsService` will be populated with at least 5 users when the application runs with the `userdetails` profile.

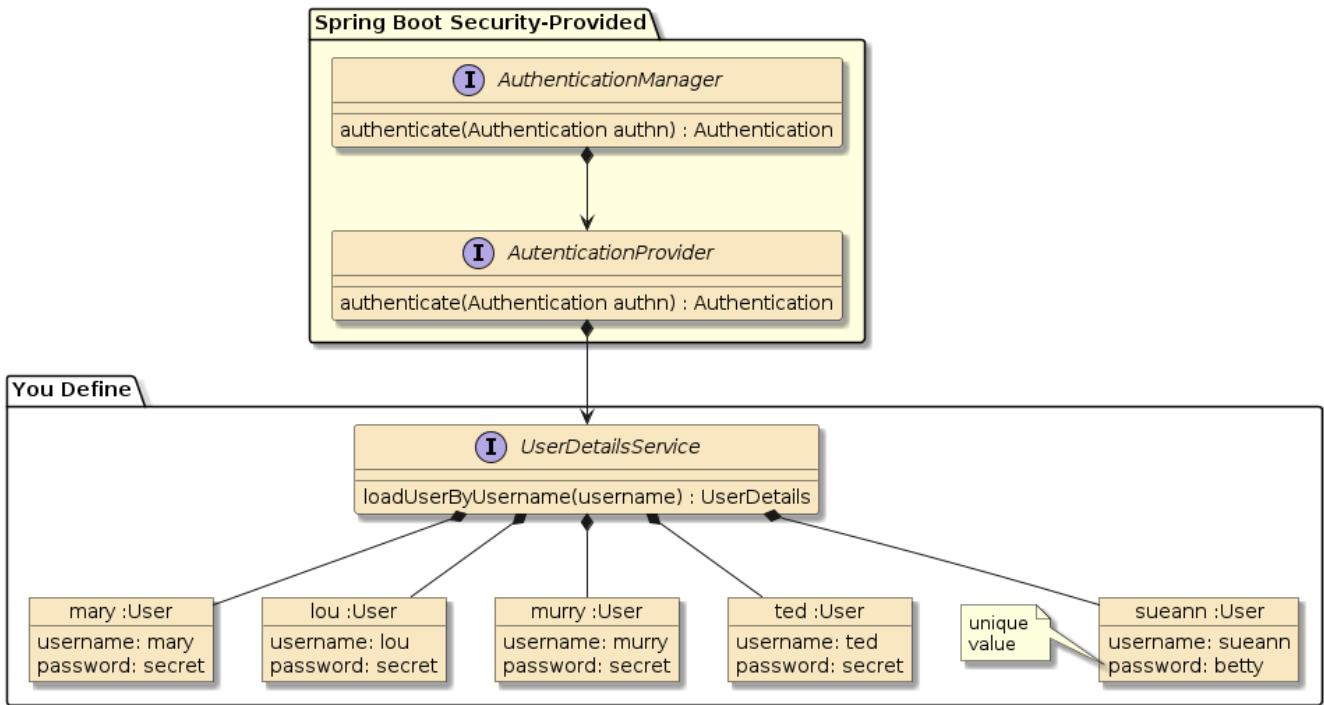


Figure 103. User Details

The `autorenters-support-security-svc` module contains a YAML file activated with the `userdetails` profile. The YAML file expresses the following users with credentials. These are made available by injecting the `Accounts @ConfigurationProperty` bean.

1. mary/secret
2. lou/secret
3. murry/secret
4. ted/secret
5. sueann/betty

Your main focus will be to provide a `UserDetailsService` that is populated from the supplied `rentalAccounts`. The account information will have only username and password.

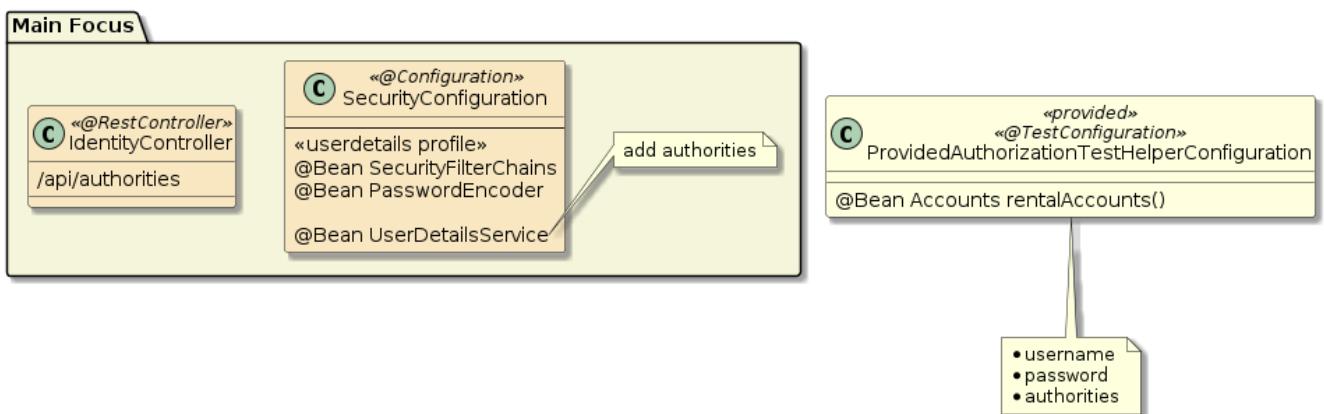


Figure 104. UserDetailsService

238.3.3. Requirements

1. Create a `UserDetailsService` that is activated during the `userdetails` profile.



The `InMemoryUserDetailsManager` is fine for this requirement. You have the option of using other implementation types if you wish but they cannot require the presence of an external resource (i.e., JDBC option must use an in-memory database).

- a. expose the `UserDetailsService` as a `@Bean` that can be injected into other factories.



```
@Bean  
public UserDetailsService userDetailsService(PasswordEncoder  
encoder, ...){
```

- b. populate it with the 5 users from an injected `Accounts @ConfigurationProperty` bean



The `rentalAccounts` bean is provided for you in the `ProvidedAuthorizationTestHelperConfiguration` in the support module.

- c. store passwords for users using a BCrypt hash algorithm.



Both the `BCryptPasswordEncoder` and the `DelegatingPasswordEncoder` will encrypt with Bcrypt.

2. Create a unit integration test case that (provided):

- a. activates the `userdetails` profile



```
@Configuration(proxyBeanMethods = false)  
@Profile({"nosecurity", "userdetails", "authorities",  
"authorization"})①  
public class PartA3_UserDetailsPart {  
====  
@SpringBootTest(classes={...},  
@ActiveProfiles({"test", "userdetails"}) ②  
public class MyA3_UserDetailsNTest extends A3_UserDetailsNTest {
```

① activated with `userdetails` (or select other) profile

② activating desired profiles

- b. verifies successful authentication and identification of the authenticated username using the `whoAmI` resource
- c. verifies successful access to a one POST/create auto, renter, and autoRental resource operation for each user



The test(s) within the support module provides much/all of this test coverage.

238.3.4. Grading

Your solution will be evaluated on:

1. build a `UserDetailsService` implementation to host user accounts and be used as a source for authenticating users
 - a. whether your solution can host credentials for multiple users
 - b. whether your tests correctly identify the authenticated caller for each of the users
 - c. whether your tests verify each authenticated user can create a Auto, Renter, and AutoRental
2. build an injectable `UserDetailsService`
 - a. whether the `UserDetailsService` was exposed using a `@Bean` factory
3. encode passwords
 - a. whether the password encoding was explicitly set to create BCrypt hash

238.3.5. Additional Details

1. There is no explicit requirement that the `UserDetailsService` be implemented using a database. If you do use a database, use an in-memory RDBMS so that there are no external resources required.
2. You may use a `DelegatingPasswordEncoder` to satisfy the BCrypt encoding requirements, but the value stored must be in BCrypt form.
3. The implementation choice for `PasswordEncoder` and `UserDetailsService` is separate from one another and can be made in separate `@Bean` factories.

```
@Bean  
public PasswordEncoder passwordEncoder() {...}  
  
@Bean  
public UserDetailsService userDetailsService(PasswordEncoder encoder, ...) {...}
```

4. The commonality of tests differentiated by different account properties is made simpler with the use of JUnit `@ParameterizedTest`. However, by default method sources are required to be declared as Java static methods—unable to directly reference beans injected into non-static attributes. `@TestInstance(TestInstance.Lifecycle.PER_CLASS)` can be used to allow the method source to be declared a Java non-static method and directly reference the injected Spring context resources.
5. You may annotate a `@Test` or `@Nested` testcase class with `@DirtiesContext` to indicate that the test makes changes that can impact other tests and the Spring context should be rebuilt after finishing.

Chapter 239. Assignment 3b: Security Authorization

239.1. Authorities

239.1.1. Purpose

In this portion of the assignment, you will demonstrate your knowledge of authorities. You will:

1. define role-based and permission-based authorities

239.1.2. Overview

In this portion of the assignment, you will start with the authorization and user details configuration of the previous section and enhance each user with authority information.

You will be assigning authorities to users and verifying them with a unit test. You will add an additional ("authorities") resource to help verify the assertions.



Figure 105. Authorities Test Resource

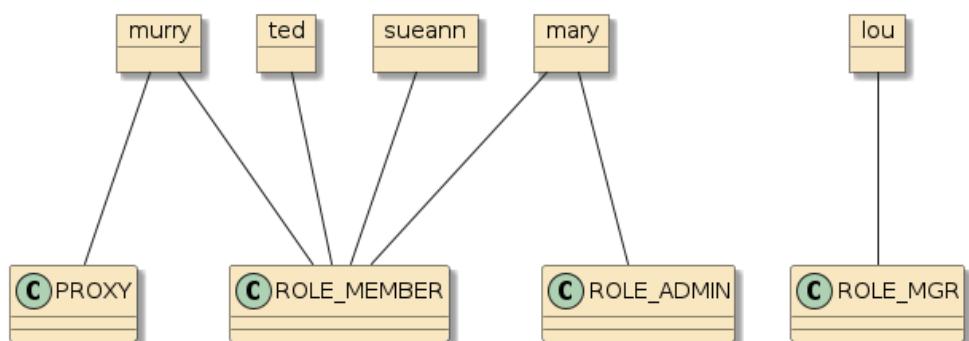


Figure 106. Assignment Principles and Authorities

The `autorenters-support-security-svc` module contains a YAML file activated with the `authorities` and `authorization` profiles. The YAML file expresses the following users, credentials, and authorities

1. mary: ROLE_ADMIN, ROLE_MEMBER
2. lou: ROLE_MGR (no role member)
3. murry: ROLE_MEMBER, PROXY
4. ted: ROLE_MEMBER
5. sueann: ROLE_MEMBER

Your focus will be on adding authorities to each user populated in the `UserDetailsService`.

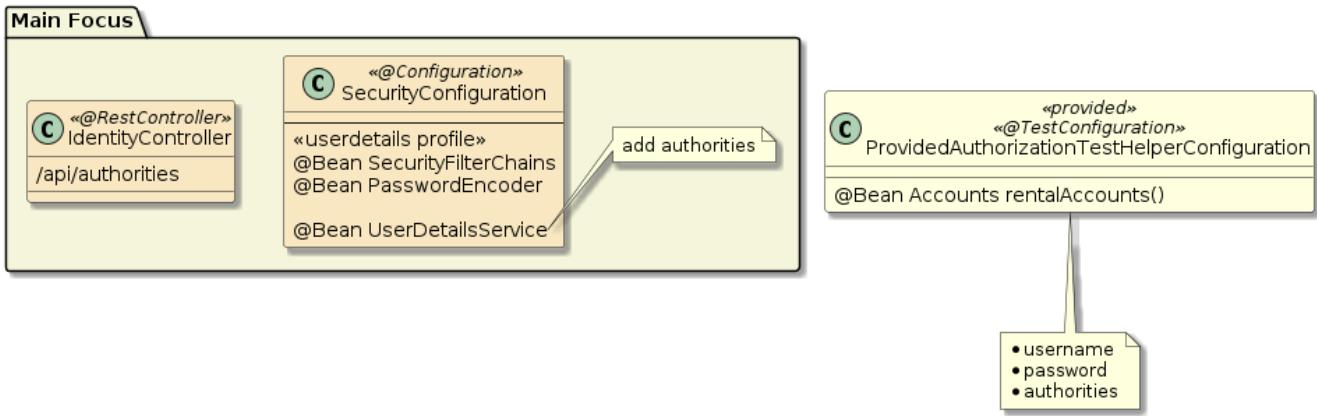


Figure 107. `UserDetailsService`

239.1.3. Requirements

1. Create an `api/authorities` resource with a `GET` method (provided):
 - a. accepts an "authority" query parameter
 - b. returns (textual) "TRUE" or "FALSE" depending on whether the caller has the authority assigned
2. Create a `UserDetailsService` that is active during the `authorities` profile.
 - a. populate it with the 5 users from an injected `Accounts @ConfigurationProperty` bean. Include pre-assigned authorities.



The `rentalAccounts` bean is provided for you in the `ProvidedAuthorizationTestHelperConfiguration` in the support module.

3. Create one or more unit integration test cases that (provided):
 - a. activates the `authorities` profile
 - b. verifies the unauthenticated caller has no authorities assigned
 - c. verifies each the authenticated callers have proper assigned authorities

239.1.4. Grading

Your solution will be evaluated on:

1. define role-based and permission-based authorities
 - a. whether you have assigned required authorities to users
 - b. whether you have verified an unauthenticated caller does not have identified authorities assigned
 - c. whether you have verified successful assignment of authorities for authenticated users as clients

239.1.5. Additional Details

1. There is no explicit requirement to use a database for the user details in this assignment.

However, if you do use an database, please use an in-memory RDBMS instance so there are no external resources required.

2. The repeated tests due to different account data can be simplified using a `@ParameterizedTest`. However, you will need to make use of `@TestInstance(TestInstanceLifecycle.PER_CLASS)` in order to leverage the Spring context in the `@MethodSource`.

239.2. Authorization

239.2.1. Purpose

In this portion of the assignment, you will demonstrate your knowledge of implementing access control based on authorities assigned for URI paths and methods. You will:

1. implement URI path-based authorization constraints
2. implement annotation-based authorization constraints
3. implement role inheritance
4. implement an `AccessDeniedException` controller advice to hide necessary stack trace information and provide useful error information to the caller

239.2.2. Overview

In this portion of the assignment you will be restricting access to specific resource operations based on path and expression-based resource restrictions.

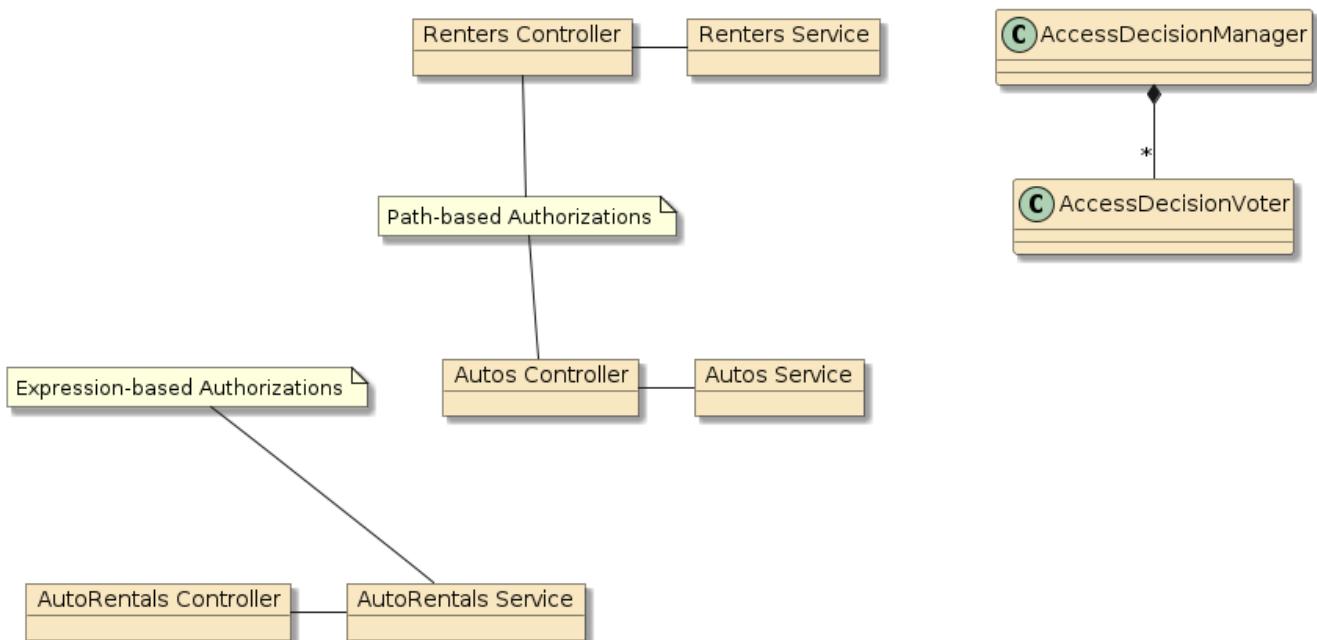


Figure 108. Authorization

Authorization Testing

```
@SpringBootTest(classes= {  
    AutoRentalsSecurityApp.class,  
    SecurityTestConfiguration.class},
```

```

webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
@ActiveProfiles({"test", "authorities", "authorization"})
public class MyB2_AuthorizationNTest extends B2_AuthorizationNTest {

```

239.2.3. Requirements

1. Update the resources to store the identity of the caller with the resource created to identify the owner
 - a. *Autos will store the username (as username) of the auto creator (provided)*
 - b. *Renters will store the username (as username) of the renter creator (provided)*
 - c. AutoRentals should store the username (as username) of the renter it represents (new)



The Renter creator or a user with the PROXY authority (murry) may create the AutoRental and have the username set to the Renter username. Therefore, there is a separation between "can they create?" and "who is the owner username?"

- d. the identity should not be included in the marshalled DTO returned to callers



The DTO may have this field omitted or marked transient so that the element is never included.



Identity for "can they create" comes from the SecurityContext. Identity for "who is the owner username" is derived from the renterId from AutoRentalDTO and RenterDTO from RentersService.

2. Define access constraints for resources using path and expression-based authorizations. Specific authorization restriction details are in the next requirement.
 - a. Use path-based authorizations for Auto and Renter resources, assigned to the URIs since you are not working with modifiable source code for these two resources



Configure HttpSecurity to enforce the required roles for autos and renters calls

- b. Use expression-based authorizations for autoRentals resources, applied to the service methods



Use annotations on AutoRental service methods to trigger authorization checks. Remember to enable method security for the `prePostEnabled` annotation form (now a default) for your application and tests.

3. Restrict access according to the following

- a. Autos (path-based authorizations)
 - i. *continue to restrict non-safe methods to authenticated users (from Authenticated Access)*

- ii. *authenticated users may modify autos that match their login (provided)*
 - iii. *authenticated users may delete autos that match their login or have the MGR role (provided)*
 - iv. only authenticated users with the **ADMIN** role can delete all autos (new)
 - b. Renters (path-based authorizations)
 - i. *continue to restrict non-safe methods to authenticated users (from Authenticated Access)*
 - ii. *authenticated users may create a single Renter to represent themselves (provided)*
 - iii. *authenticated users may modify a renter that matches their login (provided)*
 - iv. *authenticated users may delete a renter that matches their login or have the MGR role (provided)*
 - v. only authenticated users with the **ADMIN** role can delete all renters (new)
 - c. AutoRentals (annotation-based authorizations)—through the use of **@PreAuthorize** and programmatic checks.
 - i. authenticated users may create an AutoRental for an existing Renter matching their identity (new)
 - ii. authenticated users may update an **AutoRental** for an **AutoRental** they own (new)
 -  i.e. caller "sueann" can update (change time period) for an existing AutoRental associated with identity "sueann"
 - iii. authenticated users with **PROXY** authority may create and update an AutoRental for any **Renter** (new)
 -  i.e. caller "lou" and "murry" can create or update (change time period) any existing AutoRental for any Renter
 - iv. authenticated users may only delete a AutoRental for a Renter matching their username (new)
 - v. authenticated users with the **MGR** role may delete any AutoRental (new)
 - vi. authenticated users with the **ADMIN** role may delete all AutoRentals (new)
4. Form the following role inheritance
- a. **ROLE_ADMIN** inherits from **ROLE_MGR** so that users with **ADMIN** role will also be able to perform **MGR** role operations
 -  Register a **RoleHierarchy** relationship definition between inheriting roles.
 - 5. Implement a mechanism to hide stack trace or other details from the API caller response when an **AccessDeniedException** occurs. From this point forward—stack traces can be logged on the server-side but should not be exposed in the error payload.
 - 6. Create unit integration test(s) to demonstrate the behavior defined above

239.2.4. Grading

Your solution will be evaluated on:

1. implement URI path-based authorization constraints
 - a. whether path-based authorization constraints were properly defined and used for Auto and Renter resource URIs
2. implement annotation-based authorization constraints
 - a. whether expression-based authorization constraints were properly defined for AutoRental service methods
3. implement role inheritance
 - a. whether users with the **ADMIN** role were allowed to invoke methods constrained to the **MGR** role.
4. implement an `AccessDeniedException` controller advice to hide necessary stack trace information and provide useful error information to the caller
 - a. whether stack trace or other excessive information was hidden from the access denied caller response

239.2.5. Additional Details

1. Role inheritance can be defined using a `RoleHierarchy` bean.
2. With the requirements for fine-grain authority checks, the amount of additional declarative path-based and annotation-based access checking **will still be present**, but minimal. If a user and users with special authorizations can perform the same action, it will be understandable for declarative checks enforce authenticated and programmatic checks for special authority conditions.
3. An optional `AuthorizationHelper` has been provided and demonstrated in the Auto and Renter security wrappers. It has `get()`/`has()` inspection methods that simply return information from the `SecurityContext`. It also has `assert()` methods that throw `AccessDeniedException` for anything that fails. You may use it or implement your own inspection/assertion mechanisms if it does not meet your needs.

Chapter 240. Assignment 3c: HTTPS

240.1. HTTPS

240.1.1. Purpose

In this portion of the assignment, you will demonstrate your knowledge of protecting sensitive data exchanges with one-way HTTPS encryption. You will:

1. generate a self-signed certificate for demonstration use
2. enable HTTPS/TLS within Spring Boot
3. implement an integration unit test using HTTPS

240.1.2. Overview

In this portion of the assignment you will be configuring your server to support HTTPS only when the `https` profile is active.

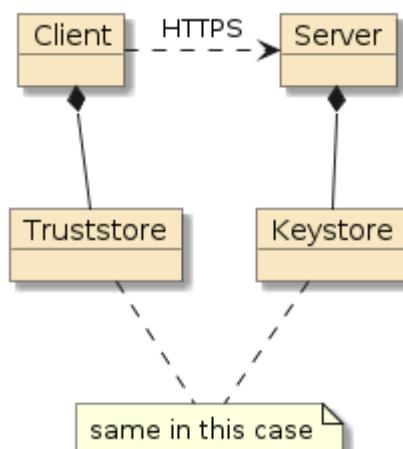


Figure 109. Authorization

240.1.3. Requirements

1. Implement an HTTPS-only communication in the server when running with the `https` profile.
 - a. package a demonstration certificate in the `src` tree
 - b. supply necessary server-side properties in a properties file

With this in place and the application started with the `authorities`, `authorization`, and `https` profiles active ...



```
java -jar target/autorentals-security-svc-1.0-SNAPSHOT-
bootexec.jar
--spring.profiles.active=authorities,authorization,https
```

should enable the following results

```

curl -k -X GET https://localhost:8443/api/whoAmI -u
"mary:secret" && echo
mary
$ curl -k -X DELETE https://localhost:8443/api/autos -u
mary:secret
$ curl -k -X DELETE https://localhost:8443/api/autos -u
sueann:betty
{"timestamp":"2022-09-
25T00:51:41.200+00:00","status":403,"error":"Forbidden","path":"/api/autos"}

```

2. Implement a unit integration test that will
 - a. activate the `authorities`, `authorization`, and `https` profiles
 - b. establish an HTTPS connection with `RestTemplate` or `WebClient`
 - c. successfully invoke using HTTPS and evaluate the result



The starter module provides a skeletal test for you to complete. The actual test performed by you can be any end-to-end communication with the server that uses HTTPS.

240.1.4. Grading

Your solution will be evaluated on:

1. generate a self-signed certificate for demonstration use
 - a. whether a demonstration PKI certificate was supplied for demonstrating the HTTPS capability of the application
2. enable HTTPS/TLS within Spring Boot
 - a. whether the integration test client was able to perform round-trip communication with the server
 - b. whether the communications used HTTPS protocol

240.1.5. Additional Details

1. There is no requirement to implement an HTTP to HTTPS redirect
2. See the [svc/svc-security/https-hello-example](#) for Maven and `RestTemplate` setup example.
3. Implement the end-to-end integration test with HTTP before switching to HTTPS to limit making too many concurrent changes.

Chapter 241. Assignment 3d: AOP and Method Proxies

In this assignment, we are going to use some cross-cutting aspects of Spring AOP and dynamic capabilities of Java reflection to modify component behavior. As a specific example, you are going to modify Auto and Renter service behavior without changing the Autos/Renters source code. We are going to add a requirement that certain fields be null and non-null.

The first two sections (reflection and dynamic proxies) of the AOP assignment lead up to the final solution (aspects) in the third section.

No Autos/Renters Compilation Dependencies



The `src/main` portions of the assignment must have no compilation dependency on Autos and Renters. All compilation dependencies and most knowledge of Autos and Renters will be in the JUnit tests.

241.1. Reflection

241.1.1. Purpose

In this portion of the assignment, you will demonstrate your knowledge of using reflection to get and invoke a method proxy. You will:

1. obtain a method reference and invoke it using Java Reflection

241.1.2. Overview

In this portion of the assignment you will implement a set of helper methods for a base class (`NullPropertyAssertion`) located in the `autorenters-support-aop` module—tasked with validating whether objects have nulls for an identified property. If the assertion fails—an exception will be thrown by the base class. Your derived class will assist in locating the method reference to the "getter" and invoking it to get the current property value.

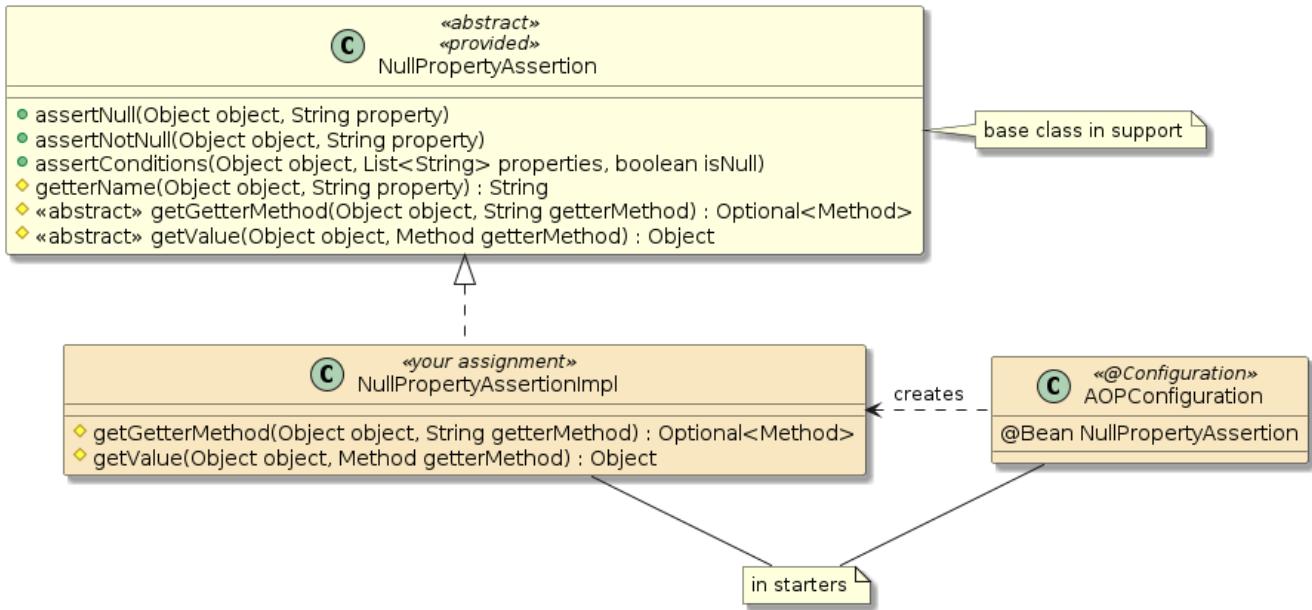


Figure 110. AOP and Method Proxies

You will find an implementation class shell, `@Configuration` class with `@Bean` factory, and JUnit test in the assignment 3 security "starter".

No Auto/Renter Compilation Dependency



Note that you see no mention of Auto or Renter in the above description/diagram. Everything will be accomplished using Java reflection.

You will need to create a dependency on the Spring Boot AOP starter, the ejava AOP support JAR, and AOP test JAR.

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
<dependency>
    <groupId>info.ejava.assignments.aop.autorentals</groupId>
    <artifactId>autorentals_homesales-support-aop</artifactId>
    <version>${ejava.version}</version>
</dependency>

<dependency>
    <groupId>info.ejava.assignments.aop.autorentals</groupId>
    <artifactId>autorentals-support-aop</artifactId>
    <classifier>tests</classifier>
    <version>${ejava.version}</version>
    <scope>test</scope>
</dependency>

```

241.1.3. Requirements

1. implement the `getGetMethod()` to locate and return the `java.lang.reflect.Method` for the

requested (getter) method name.

- a. return the `Method` object if it exists
- b. use an `Optional<Method>` return type and return an `Optional.empty()` if it does not exist. It is not an error if the getter does not exist.



It is not considered an error if the `getterName` requested does not exist. JUnit tests—which know the full context of the call—will decide if the result is correct or not.

2. implement the `getValue()` method to return the value reported by invoking the getter method using reflection
 - a. return the value returned
 - b. report any exception thrown as a runtime exception



Any exception calling an existing getter is unexpected and should be reported as a (subclass of) `RuntimeException` to the higher-level code to indicate this type of abnormality

3. use the supplied JUnit unit tests to validate your solution. There is no Spring context required/used for this test.



The tests will use non-AutoDTO/RenterDTO objects on purpose. The validator under test must work with any type of object passed in. Only "getter" method access will be supported for this capability. No "field" access will be performed.

241.1.4. Grading

Your solution will be evaluated on:

1. obtain a method reference and invoke it using Java Reflection
 - a. whether you are able to return a reference to the specified getter method using reflection
 - b. whether you are able to obtain the current state of the property using the getter method reference using reflection

241.1.5. Additional Details

1. The JUnit test (`MyD1_ReflectionMethodTest`) is supplied and should not need to be modified beyond enabling it.
2. The tests will feed your solution with DTO instances in various valid and invalid states according to the `isNull` and `notNull` calls. There will also be some non-DTO classes used to verify the logic is suitable for generic parameter types.

241.2. Dynamic Proxies

241.2.1. Purpose

In this portion of the assignment, you will demonstrate your knowledge of implementing a dynamic proxy. You will:

1. create a JDK Dynamic Proxy to implement adhoc interface(s) to form a proxy at runtime for implementing advice

241.2.2. Overview

In this portion of the assignment you will implement a dynamic proxy that will invoke the `NullPropertyAssertion` from the previous part of the assignment. The primary work will be in implementing an `InvocationHandler` implementation that will provide the implementation "advice" to the target object for selected methods. The advice will be a null check of specific properties of objects passed as parameters to the target object.

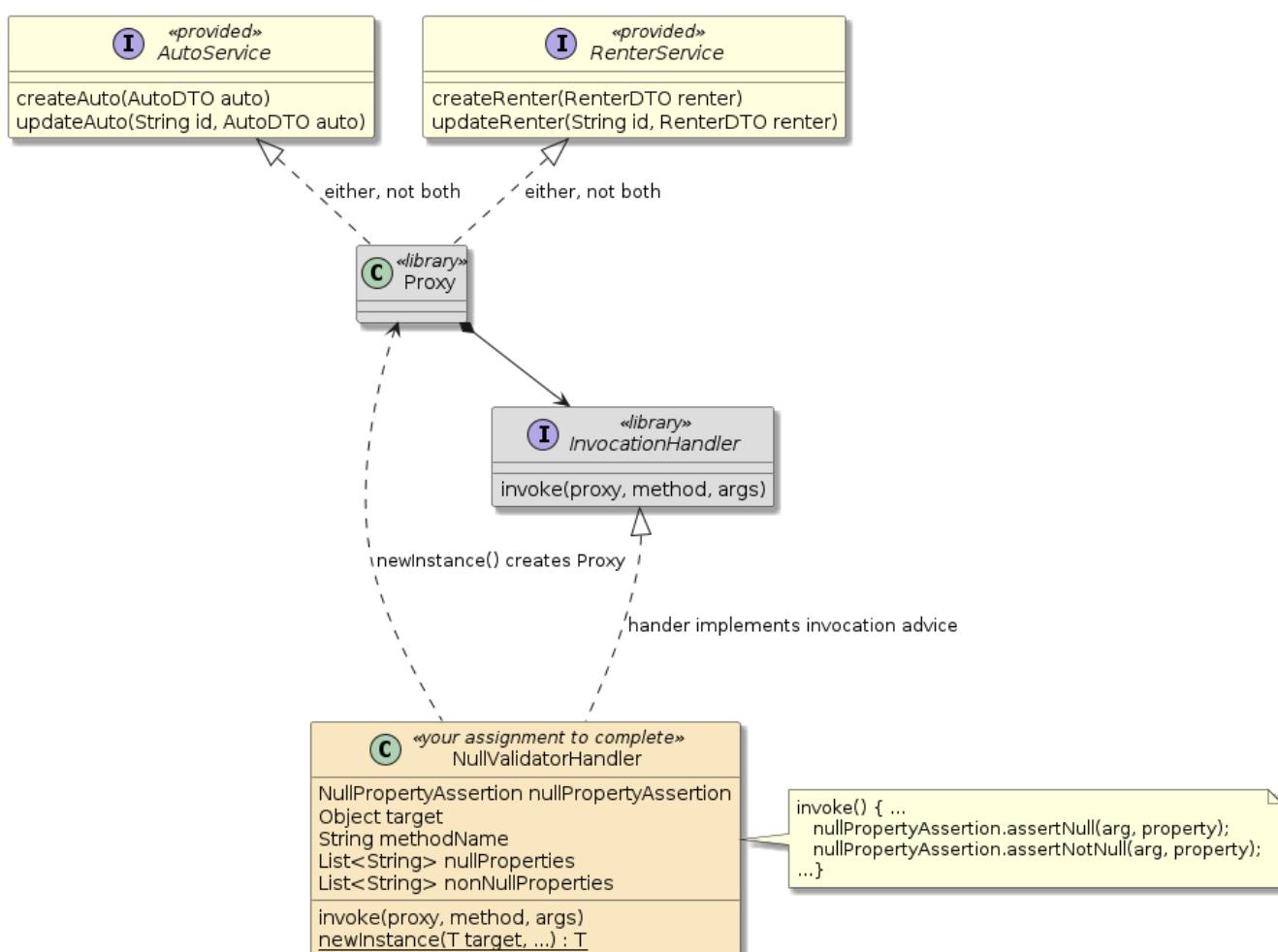


Figure 111. Dynamic Proxy Advice

The constructed `Proxy` will be used as a stepping stone to better understand the Aspect solution you will implement in the next section. The handler/proxy will not be used in the final solution. It will be instantiated and tested within the JUnit test using an injected `AutoService` and `RentersService` from the Spring context.



No AutoDTO/RenterDTO Compilation Dependency

There will be no mention of or direct dependency on Autos or Renters in your

solution. Everything will be accomplished using Java reflection.

241.2.3. Requirements

1. implement the details for the `NullValidatorHandler` class that
 - a. implements the `java.lang.reflect.InvocationHandler` interface
 - b. has implementation properties
 - i. `nullPropertyAssertion` — implements the check (from the previous section)
 - ii. target object that it is the proxy for
 - iii. `methodName` it will operate against on the target object
 - iv. `nullProperties` — `propertyNames` it will test for null (used also in the previous section)
 - v. `nonNullProperties` — `propertyNames` it will test for non-null (used also in the previous section)
 - c. has a static builder method called `newInstance` that accepts the above properties and returns a `java.lang.reflect.Proxy` implementing all interfaces of the target

In order, for the tests to locate your factory method, it must have the following exact signature.



```
//NullPropertyAssertion.class, Object.class, String.class,
List.class, List.class
public static <T> T newInstance(
    NullPropertyAssertion nullPropertyAssertion,
    T target,
    String methodName,
    List<String> nullProperties,
    List<String> nonNullProperties) {
```



`org.apache.commons.lang3.ClassUtils` can be used to locate all interfaces of a class.

- d. implements the `invoke()` method of the `InvocationHandler` interface to validate the arguments passed to the method matching the `methodName`. Checks properties matching `nullProperties` and `nonNullProperties`.

Example Validation Context

- 
- Example: `renterService.createRenter(renterDTO)`
 - **target** - `renterService`
 - **method** - `createRenter`
 - **arg** - `renterDTO`
 - **id** - property to evaluate

- Example: autoService.getAuto(autoId)

- **target** - autoService
- **method** - getRenter
- **arg** - autoId

Test Method Invocations Matching methodName for this Instance

Test only methods that match the configured method name for the proxy instance.



```
private final String methodName;

public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
    if (this.methodName.equals(method.getName())) {
```

Test Method Argument Properties not Target Properties

Test the properties of each argument — not the properties of the target. Use null and non-null property lists provided during construction.



```
nullPropertyAssertion.assertConditions(arg, isNullProperties,
true);
nullPropertyAssertion.assertConditions(arg, nonNullProperties,
false);
```

- i. if the arguments are found to be in error — let the nullPropertyAssertion throw its exception
- ii. otherwise allow the call to continue to the target object/method with provided args

Validator limits



Each proxy instance will only validate parameters of the named method but all parameters to that method. There is no way to distinguish between param1.propertyName and param2.propertyName with our design and will not need to

2. Use the provided JUnit tests ([MyD2_DynamnicProxyNTest](#)) verify the functionality of the requirements above once you activate.
 - a. Provide a NullPropertyAssertion component in the Spring context (e.g., `@Bean` factory)



Make sure the `@Bean` factory is in the component scan path of your `@SpringBootApplication` application class.

241.2.4. Grading

Your solution will be evaluated on:

1. create a JDK Dynamic Proxy to implement adhoc interface(s) to form a proxy at runtime for implementing advice
 - a. whether you created a `java.lang.reflect.InvocationHandler` that would perform the validation on proxied targets
 - b. whether you successfully created and returned a dynamic proxy from the `newInstance` factory method
 - c. whether your returned proxy was able to successfully validate arguments passed to target

241.2.5. Additional Details

1. The JUnit test case uses real AutoService and RenterService @Autowired from the Spring context, but only instantiates the proxy as a POJO within the test case.
2. The JUnit tests make calls to the AutoService and RenterService, passing valid and invalid instances according to how your proxy was instantiated during the call to `newInstance()`.
3. The completed dynamic proxy will not be used beyond this section of the assignment. However, try to spot what the dynamic proxy has in common with the follow-on Aspect solution—since Spring interface Aspects are implemented using Dynamic Proxies.

241.3. Aspects

241.3.1. Purpose

In this portion of the assignment, you will demonstrate your knowledge of adding functionality to a point in code using an Aspect. You will:

1. implement dynamically assigned behavior to methods using Spring Aspect-Oriented Programming (AOP) and AspectJ
2. identify method join points to inject using pointcut expressions
3. implement advice that executes before join points
4. implement parameter injection into advice

241.3.2. Overview

In this portion of the assignment you will implement a `ValidatorAspect` that will "advise" service calls to provide secure Auto/Renter wrapper services.

The aspect will be part of your overall Spring context and will be able to change the behavior of the Autos and Renters used within your runtime application and tests when activated. The aspect will specifically reject any object passed to these services that violate defined create/update constraints. The aspect will be defined to match the targeted methods and arguments but will have no compilation dependency that is specific to Autos or Renters.

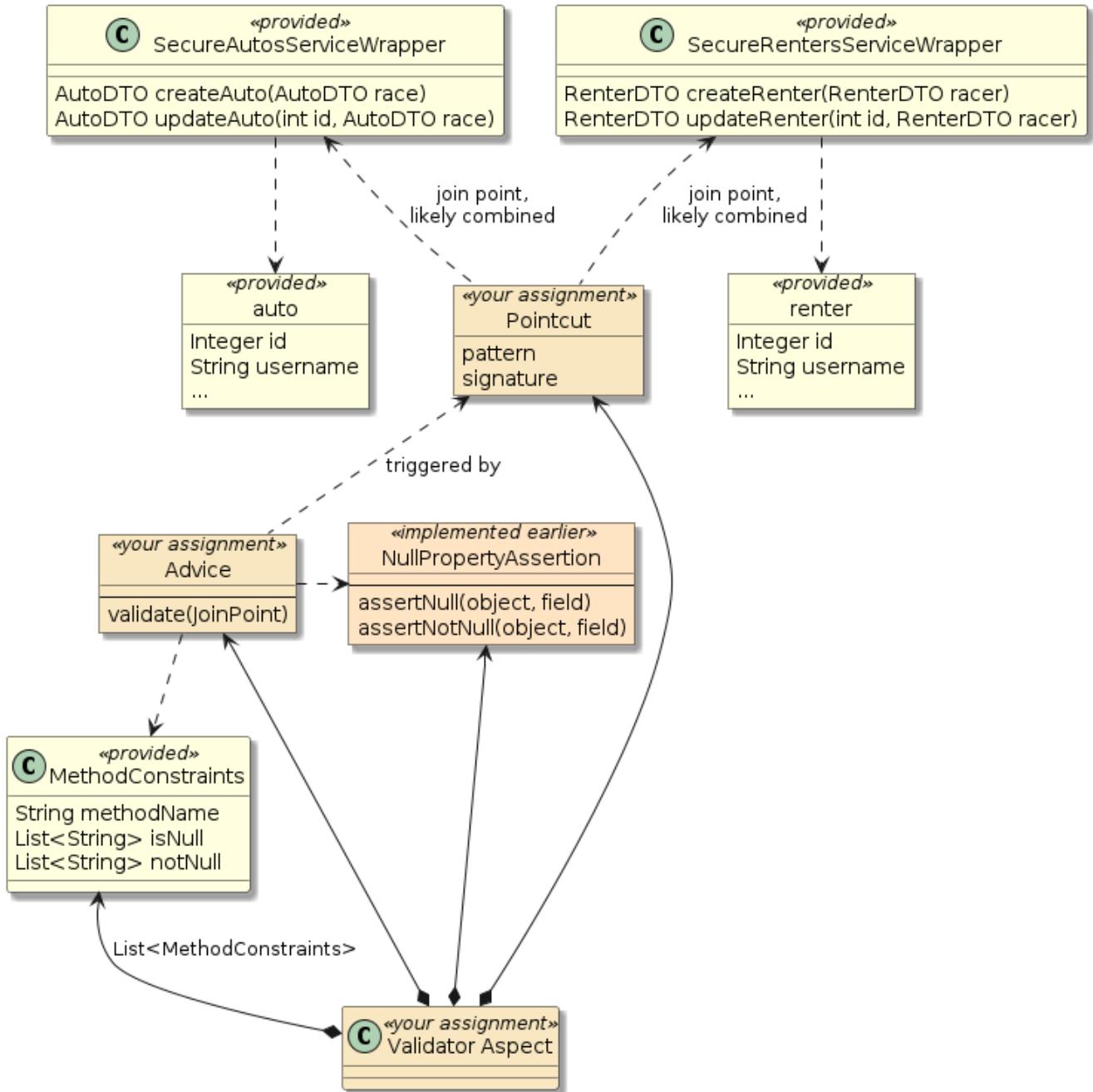


Figure 112. Aspect Advice



No AutoDTO/RenterDTO Compilation Dependency

There will be no direct dependency on Autos or Renters. Everything will be achieved using AOP expressions and Java reflection constructs.

Your primary focus in the starter will be the following classes:

- AOPConfiguration
- ValidatorAspect

241.3.3. Requirements

1. add AOP dependencies using the `spring-boot-starter-aop` module (provided)
2. enable AspectJ auto proxy handling within your application

3. create a `ValidatorAspect` component. The shell of a class and `@Bean` factory have been provided.
 - a. inject a `NullPropertyAssertion` bean (implemented in the previous section) and `List<MethodConstraints>` (populated from application-aop.yaml from the support module; provided)
 - i. `NullPropertyAssertion @Bean` factory should already be in place from the previous section
 - ii. `List<MethodConstraints> @Bean` factory is provided for you in the starter module
 - b. annotate the Aspect class that will contain the pointcut(s) and advice with "`@Aspect`"
 - c. make the overall component conditional on the `aop` profile



This means the Auto and Renter services will act as delivered when the `aop` profile is not active and enforce the new constraints when the profile is active.

4. define a "pointcut" that will target the calls to the `SecureAutosWrapper` and `SecureRentersWrapper` services. This pointcut should both:
 - a. define a match pattern for the "join point"



Start with Lenient Pattern

Start with a lenient pattern for initial success and look to optimize with more precise matching over time.

5. define an "advice" method to execute before the "join point"
 - a. uses the previously defined "pointcut" to identify its "join point"
 - b. uses typed or dynamic advice parameters



Using typed advice parameters will require a close relationship between advice and service methods. Using dynamic advice parameters (`JoinPoint`) will allow a single advice method be used for all service methods. The former is more appropriate for targeted behavior. The latter is more appropriate to general purpose behavior.

- c. invokes the `NullPropertyAssertion` bean with the parameters passed to the method and asks to validate for `isNull` and `notNull`.



```
nullPropertyAssertion.assertConditions(arg, conditions.  
getIsNull(), true);  
nullPropertyAssertion.assertConditions(arg, conditions  
.getNotNull(), false);
```

6. Use the provided JUnit test cases to verify completion of the above requirements
 - a. the initial `MyD3a1_NoAspectSvcNTest` deactivates the `aop` profile to demonstrate baseline behavior we want to change/not-change

- b. the second `MyD3a2_AspectSvcNTest` activates the `aop` profile and asserts different results
- c. the third `MyD3b_AspectWebNTest` demonstrates that the aspect was added to the beans injected into the Autos and Renters controllers.



If `AspectSvcNTest` passes, and `AspectWebNTest` fails, check that you are advising the secure wrapper and not the base service.

241.3.4. Grading

Your solution will be evaluated on:

1. implement dynamically assigned behavior to methods using Spring Aspect-Oriented Programming (AOP) and AspectJ
 - a. whether you activated aspects in your solution
 - b. whether you supplied an aspect as a component
2. identify method join points to inject using pointcut expressions
 - a. whether your aspect class contained a pointcut that correctly matched the target method join points
3. implement advice that executes before join points
 - a. whether your solution implements required validation before allowing target to execute
 - b. whether your solution will allow the target to execute if validation passes
 - c. whether your solution will prevent the target from executing and report the error if validation fails
4. implement parameter injection into advice
 - a. whether you have implemented typed or dynamic access to the arguments passed to the target method

241.3.5. Additional Details

1. You are to base your AOP validation based on the data found within the injected `List<MethodConstraints>`. Each instance contains the name of the method and a list of property names that should be subject to `isNull` or `notNull` assertions.

application-aop.yaml (in support)

```
aop:
  validation:
    - methodName: createAuto
      isNull: [id, username]
      notNull: [make, model, passengers, dailyRate, fuelType ]
    - methodName: updateAuto
      isNull: [username]

    - methodName: createRenter
      isNull: [id, username]
```

```
notNull: [make, firstName, lastName, dob, email]
- methodName: updateRenter
isNull: [username]
```

2. The JUnit test case uses real, secured AutosService and RenterServices @Autowired from the Spring context augmented with aspects. Your aspect will be included when the `aop` profile is active.
3. The JUnit tests will invoke the security service wrappers directly and through the controllers—calling with valid and invalid parameters according to the definition in the `application-aop.yaml` file.



The `Auto` and `Renter` services may have some base validation built in and will not accept a non-null or null values. You cannot change that behavior and will not have to. You will validate the properties as defined, coming into the service.

4. Ungraded activity—create a breakpoint in the Advice and Autos/RentersService(s) when executing the tests. Observe the call stack to see how you got to that point, where you are headed, and what else is in that call stack.

Spring AOP and Method Proxies

copyright © 2024 jim stafford (jim.stafford@jhu.edu)

Chapter 242. Introduction

Many times, business logic must execute additional behavior outside its core focus. For example, auditing, performance metrics, transaction control, retry logic, etc. We need a way to bolt on additional functionality ("advice") without knowing what the implementation code ("target") will be, what interfaces it will implement, or even if it will implement an interface.

Frameworks must solve this problem every day. To fully make use of advanced frameworks like Spring and Spring Boot, it is good to understand and be able to implement solutions using some of the dynamic behavior available like:

- Java Reflection
- Dynamic (Interface) Proxies
- CGLIB (Class) Proxies
- Aspect Oriented Programming (AOP)

242.1. Goals

You will learn:

- to decouple potentially cross-cutting logic away from core business code
- to obtain and invoke a method reference
- to wrap add-on behavior to targets in advice
- to construct and invoke a proxy object containing a target reference and decoupled advice
- to locate callable join point methods in a target object and apply advice at those locations
- to enhance services and objects with additional state and behavior

242.2. Objectives

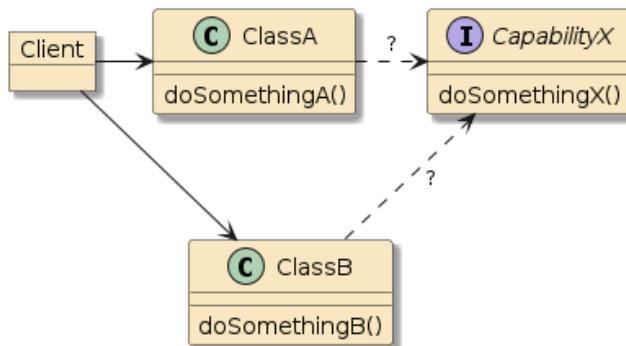
At the conclusion of this lecture and related exercises, you will be able to:

1. obtain a method reference and invoke it using Java Reflection
2. create a JDK Dynamic Proxy to implement adhoc interface(s) to form a proxy at runtime for implementing advice
3. create a CGLIB Proxy to dynamically create a subclass to form a proxy at runtime for implementing advice
4. create an AOP proxy programmatically using a [ProxyFactory](#)
5. implement dynamically assigned behavior to components in the Spring context using Spring Aspect-Oriented Programming (AOP) and AspectJ
6. identify method join points to inject using pointcut expressions
7. implement advice that executes before, after, and around join points
8. implement parameter injection into advice

9. enhance services and objects at runtime with additional state using Introductions

Chapter 243. Rationale

Our problem starts off with two independent classes depicted as `ClassA` and `ClassB` and a caller labeled as `Client`. `doSomethingA()` is unrelated to `doSomethingB()` but may share some current or future things in common—like transactions, database connection, or auditing requirements.



We come to a point where `CapabilityX` is needed in both `doSomethingA()` and `doSomethingB()`. An example of this could be normalization or input validation. We could implement the capability within both operations or in near-best situations implement a common solution and have both operations call that common code.

Reuse is good, but depending on how you reuse, it may be more intrusive than necessary.

Figure 113. New Cross-Cutting Design Decision

243.1. Adding More Cross-Cutting Capabilities

Of course, it does not end there, and we have established what could be a bad pattern of coupling the core business code of `doSomethingA()` and `doSomethingB()` with tangential features of the additional capabilities (e.g., auditing, timing, retry logic, etc.).

What other choice do we have?

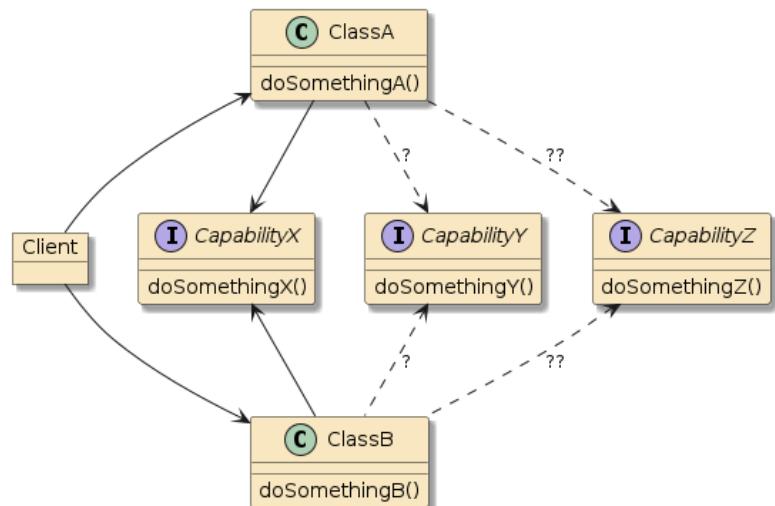
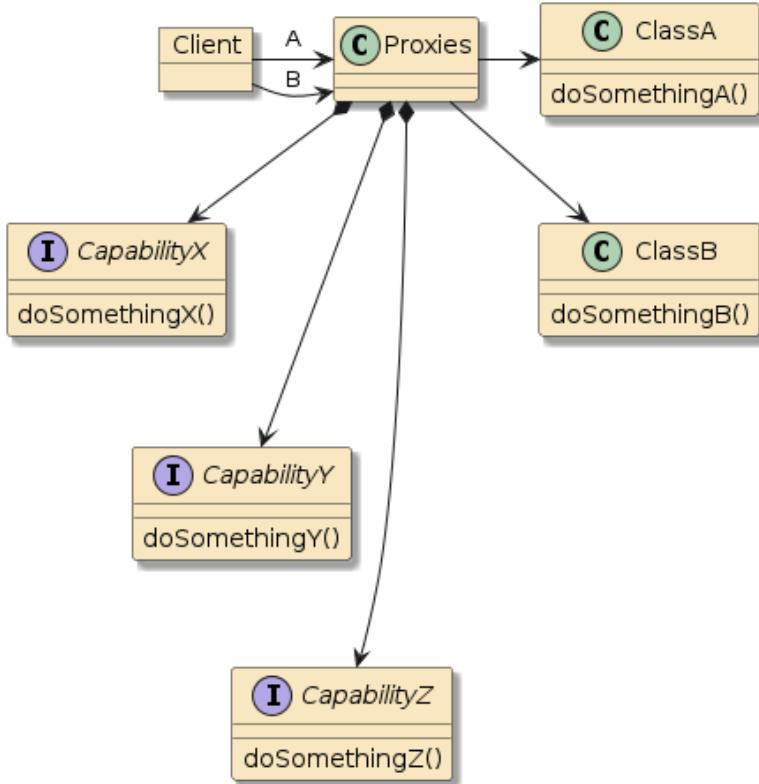


Figure 114. More Cross-Cutting Capabilities

243.2. Using Proxies



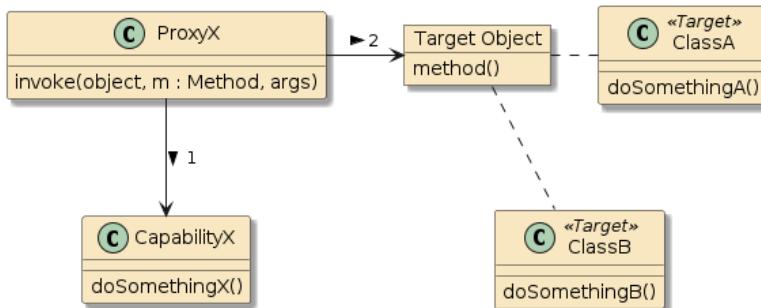
What we can do instead is leave `ClassA` and `ClassB` alone and wrap calls to them in a series of one or more proxied calls. `X` might perform auditing, `Y` might perform normalization of inputs, and `Z` might make sure the connection to the database is available and a transaction active. All we need `ClassA` and `ClassB` to do is their designated "something".

However, there is a slight flaw to overcome. `ClassA` and `ClassB` do not implement a common interface; `doSomethingA()` and `doSomethingB()` look very different in signature, and capabilities; neither `X`, `Y`, `Z` or any of the proxy layers know a thing about `ClassA` or `ClassB`.

We need to tie these unrelated parts together. Let's begin to solve this with Java Reflection.

Chapter 244. Reflection

Java Reflection provides a means to examine a Java class and determine facts about it that can be useful in describing it and invoking it.



Let's say I am in **ProxyX** applying **doSomethingX()** to the call and I want to invoke some to-be-determined (TBD) method in the target object. **ProxyX** does not need to know anything except what to call to have the target invoked. This call will eventually point to **doSomethingA()** or **doSomethingB()** at some point.

We can use Java Reflection to solve this problem by

- inspecting the target object's class (**ClassA** or **ClassB**) to obtain a reference to the method (**doSomethingA()** or **doSomethingB()**) we wish to call
- identify the arguments to be passed to the call
- identify the target object to call

Let's take a look at this in action.

244.1. Reflection Method

Java Reflection provides the means to get access to **Fields** and **Methods** of a class. In the example below, I show code that gets a reference to the **createItem** method, in the **ItemsService** interface, and accepts an object of type **ItemDTO**.

Obtaining a Java Reflection Method Reference

```
import info.ejava.examples.svc.aop.items.services.ItemsService;
import java.lang.reflect.Method;
...
Method method = ItemsService.class.getMethod("createItem", ItemDTO.class); ①
log.info("method: {}", method);
...
```

① getting reference to method within **ItemsService** interface

Java Class has numerous methods that allow us to inspect interfaces and classes for fields, methods, annotations, and related types (e.g., inheritance). **getMethod()** looks for a method with the String name ("createItem") that accepts the supplied type(s) (**ItemDTO**). Arguments is a vararg array, so we can pass in as many types as necessary to match the intended call.

The result is a **Method** instance that we can use to refer to the specific method to be called—but not the target object or specific argument values.

Example Reflection Method Output

```
method: public abstract info.ejava.examples.svc.aop.items.dto.ItemDTO  
    info.ejava.examples.svc.aop.items.services.ItemsService.createItem(  
        info.ejava.examples.svc.aop.items.dto.ItemDTO)
```

244.2. Calling Reflection Method

We can invoke the **Method** reference with a target object and arguments and receive the response as a **java.lang.Object**.

Example Reflection Method Call

```
import info.ejava.examples.svc.aop.items.dto.BedDTO;  
import info.ejava.examples.svc.aop.items.services.ItemsService;  
import java.lang.reflect.Method;  
  
...  
  
ItemsService<BedDTO> bedsService = ... ①  
Method method = ... ②  
  
//invoke method using target object and args  
Object[] args = new Object[] { BedDTO.bedBuilder().name("Bunk Bed").build() }; ③  
log.info("invoke calling: {}({})", method.getName(), args);  
  
Object result = method.invoke(bedsService, args); ④  
  
log.info("invoke {} returned: {}", method.getName(), result);
```

- ① obtain a target object to invoke
- ② get a Method reference like what was shown earlier
- ③ arguments are passed into **invoke()** using a varargs array
- ④ invoke the method on the object and obtain the result

Example Method Reflection Call Output

```
invoke calling: createItem([BedDTO(super=ItemDTO(id=0, name=Bunk Bed))])  
invoke createItem returned: BedDTO(super=ItemDTO(id=1, name=Bunk Bed))
```

244.3. Reflection Method Result

The end result is the same as if we called the **BedsServiceImpl** directly.

Example Method Reflection Result

```
//obtain result from invoke() return  
BedDTO createdBed = (BedDTO) result;
```

```
log.info("created bed: {}", createdBed);----
```

Example Method Reflection Result Output

```
created bed: BedDTO(super=ItemDTO(id=1, name=Bunk Bed))
```

There, of course, is more to Java Reflection that can fit into a single example—but let's now take that fundamental knowledge of a [Method](#) reference and use that to form some more encapsulated proxies using JDK Dynamic (Interface) Proxies and CGLIB (Class) Proxies.

Chapter 245. JDK Dynamic Proxies

The JDK offers a built-in mechanism for creating dynamic proxies for interfaces. These are dynamically generated classes—when instantiated at runtime—will be assigned an arbitrary set of interfaces to implement. This allows the generated proxy class instances to be passed around in the application, masquerading as the type(s) they are a proxy for. This is useful in frameworks to implement features for implementation types they will have no knowledge of until runtime. This eliminates the need for compile-time generated proxies.^[1]

245.1. Creating Dynamic Proxy

We create a JDK Dynamic Proxy using the static `newProxyInstance()` method of the `java.lang.reflect.Proxy` class. It takes three arguments:

- the classloader for the supplied interfaces
- the interfaces to implement
- the handler to implement the custom advice details of the proxy code and optionally complete the intended call (e.g., security policy check handler)

In the example below, `GrillServiceImpl` extends `ItemsServiceImpl<T>`, which implements `ItemsService<T>`. We are creating a dynamic proxy that will implement that interface and delegate to an advice instance of `MyInvocationHandler` that we write.

Creating Dynamic Proxy

```
import info.ejava.examples.svc.aop.items.aspects.MyDynamicProxy;
import info.ejava.examples.svc.aop.items.services.GrillsServiceImpl;
import info.ejava.examples.svc.aop.items.services.ItemsService;
import java.lang.reflect.Proxy;
...

ItemsService<GrillDTO> grillService = new GrillsServiceImpl(); ①

ItemsService<GrillDTO> grillServiceProxy = (ItemsService<GrillDTO>)
    Proxy.newProxyInstance( ②
        grillService.getClass().getClassLoader(),
        new Class[]{ItemsService.class}, ③
        new MyInvocationHandler(grillService) ④
    );
log.info("created proxy {}", grillServiceProxy.getClass());
log.info("handler: {}",
    Proxy.getInvocationHandler(grillServiceProxy).getClass());
log.info("proxy implements interfaces: {}",
    ClassUtils.getAllInterfaces(grillsServiceProxy.getClass()));
```

① create target implementation object unknown to dynamic proxy

- ② instantiate dynamic proxy instance and underlying dynamic proxy class
- ③ identify the interfaces implemented by the dynamic proxy class
- ④ provide advice instance that will handle adding proxy behavior and invoking target instance

245.2. Generated Dynamic Proxy Class Output

The output below shows the `$Proxy86` class that was dynamically created and that it implements the `ItemsService` interface and will delegate to our custom `MyInvocationHandler` advice.

Example Generated Dynamic Proxy Class Output

```
created proxy: class com.sun.proxy.$Proxy86
handler: class info.ejava.examples.svc.aop.items.aspects.MyInvocationHandler
proxy implements interfaces:
[interface info.ejava.examples.svc.aop.items.services.ItemsService, ①
 interface java.io.Serializable] ②
```

① `ItemService` interface supplied at runtime

② `Serializable` interface implemented by DynamicProxy implementation class

245.3. Alternative Proxy All Construction

Alternatively, we can write a convenience builder that simply forms a proxy for all implemented interfaces of the target instance. The [Apache Commons ClassUtils](#) utility class is used to obtain a list of all interfaces implemented by the target object's class and parent classes.

Alternative Proxy All Construction

```
import org.apache.commons.lang3.ClassUtils;
...
@RequiredArgsConstructor
public class MyInvocationHandler implements InvocationHandler {
    private final Object target;

    public static Object newInstance(Object target) {
        return Proxy.newProxyInstance(target.getClass().getClassLoader(),
            ClassUtils.getAllInterfaces(target.getClass()).toArray(new Class[0]), ①
            new MyInvocationHandler(target));
    }
}
```

① [Apache Commons ClassUtils](#) used to obtain all interfaces for target object

245.4. InvocationHandler Class

JDK Dynamic Proxies require an instance that implements the `InvocationHandler` interface to implement the custom work (aka "advice") and delegate the call to the target instance (aka "around advice"). This is a class that we write. The `InvocationHandler` interface defines a single reflection-

oriented `invoke()` method taking the proxy, method, and arguments to the call. Construction of this object is up to us—but the raw target object is likely a minimum requirement—as we will need that to make a clean, delegated call.

Example Dynamic Proxy InvocationHandler

```
...
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
...
@RequiredArgsConstructor
public class MyInvocationHandler implements InvocationHandler { ①
    private final Object target; ②

    @Override
    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable { ③
        //proxy call
    }
}
```

① class must implement `InvocationHandler`

② raw target object to invoke

③ `invoke()` is provided reflection information for call

245.5. `InvocationHandler` `invoke()` Method

The `invoke()` method performs any necessary advice before or after the proxied call and uses standard method reflection to invoke the target method. This is the same `Method` class from the earlier discussion on Java Reflection. The response or thrown exception can be directly returned or thrown from this method.

InvocationHandler Proxy invoke() Method

```
@Override
public Object invoke(Object proxy, Method method, Object[] args)
    throws Throwable {
    //do work ...
    log.info("invoke calling: {}({})", method.getName(), args);

    Object result = method.invoke(target, args);

    //do work ...
    log.info("invoke {} returned: {}", method.getName(), result);
    return result;
}
```



Must invoke raw target instance—not the proxy

Calling the supplied proxy instance versus the raw target instance would result in a circular loop. We must somehow have a reference to the raw target to be able to directly invoke that instance.

245.6. Calling Proxied Object

The following is an example of the proxied object being called using its implemented interface.

Example Proxied Object Call

```
GrillDTO createdGrill = grillServiceProxy.createItem(  
    GrillDTO.grillBuilder().name("Broil King").build());  
log.info("created grill: {}", createdGrill);
```

The following shows that the call was made to the target object, work was able to be performed before and after the call within the `InvocationHandler`, and the result was passed back as the result of the proxy.

Example Proxied Call Output

```
invoke calling: createItem([GrillDTO(super=ItemDTO(id=0, name=Broil King))]) ①  
invoke createItem returned: GrillDTO(super=ItemDTO(id=1, name=Broil King)) ②  
created grill: GrillDTO(super=ItemDTO(id=1, name=Broil King)) ③
```

- ① work performed within the `InvocationHandler` advice prior to calling target
- ② work performed within the `InvocationHandler` advice after calling target
- ③ target method's response returned to proxy caller

JDK Dynamic Proxies are definitely a level up from constructing and calling `Method` directly as we did with straight Java Reflection. They are the proxy type of choice within Spring but have the limitation that they can only be used to proxy interface-based objects and not no-interface classes.

If we need to proxy a class that does not implement an interface — CGLIB is an option.

[1] "Dynamic Proxy Classes", Oracle JavaSE 8 Technotes

Chapter 246. CGLIB

Code Generation Library (CGLIB) is a byte instrumentation library that allows the manipulation or creation of classes at runtime.^[1]

Where JDK Dynamic Proxies implement a proxy behind an interface, CGLIB dynamically implements a subclass of the class proxied.

This library has been fully integrated into [spring-core](#), so there is nothing additional to add to begin using it directly (and indirectly when we get to Spring AOP).

246.1. Creating CGLIB Proxy

The following code snippet shows a CGLIB proxy being constructed for a [ChairsServiceImpl](#) class that implements no interfaces. Take note that there is no separate target instance — our generated proxy class will be a subclass of [ChairsServiceImpl](#) and it will be part of the target instance. The real target will be in the base class of the instance. We register an instance of [MethodInterceptor](#) to handle the custom advice and optionally complete the call. This is a class that we write when authoring CGLIB proxies.

Creating CGLIB Proxy

```
import info.ejava.examples.svc.aop.items.aspects.MyMethodInterceptor;
import info.ejava.examples.svc.aop.items.services.ChairsServiceImpl;
import org.springframework.cglib.proxy.Enhancer;
...
Enhancer enhancer = new Enhancer();
enhancer.setSuperclass(ChairsServiceImpl.class); ①
enhancer.setCallback(new MyMethodInterceptor()); ②
ChairsServiceImpl chairsServiceProxy = (ChairsServiceImpl)enhancer.create(); ③

log.info("created proxy: {}", chairsServiceProxy.getClass());
log.info("proxy implements interfaces: {}",
    ClassUtils.getAllInterfaces(chairsServiceProxy.getClass()));
```

- ① create CGLIB proxy as subclass of target class
- ② provide instance that will handle adding proxy advice behavior and invoking base class
- ③ instantiate CGLIB proxy — this is our target object

The following output shows that the proxy class is of a CGLIB proxy type and implements no known interface other than the CGLIB [Factory](#) interface. Note that we were able to successfully cast this proxy to the [ChairsServiceImpl](#) type — the assigned base class of the dynamically built proxy class.

Example Generated CGLIB Proxy Class

```
created proxy: class
info.ejava.examples.svc.aop.items.services.GrillsServiceImpl$$EnhancerByCGLIB$$a4035db
5
```

proxy implements interfaces: [interface org.springframework.cglib.proxy.Factory] ①

① Factory interface implemented by CGLIB proxy implementation class

246.2. MethodInterceptor Class

To intelligently process CGLIB callbacks, we need to supply an advice class that implements `MethodInterceptor`. This gives us access to the proxy instance being invoked, the reflection `Method` reference, call arguments, and a new type of parameter — `MethodProxy`, which is a reference to the target method implementation in the base class.

Example MethodInterceptor Class

```
...
import org.springframework.cglib.proxy.MethodInterceptor;
import org.springframework.cglib.proxy.MethodProxy;
import java.lang.reflect.Method;

public class MyMethodInterceptor implements MethodInterceptor {
    @Override
    public Object intercept(Object proxy, Method method, Object[] args,
                           MethodProxy methodProxy) ①
        throws Throwable {
        //proxy call
    }
}
```

① additional method used to invoke target object implementation in base class

246.3. MethodInterceptor intercept() Method

The details of the `intercept()` method are much like the other proxy techniques we have looked at and will look at in the future. The method has a chance to do work before and after calling the target method, optionally calls the target method, and returns the result. The main difference is that this proxy is operating within a subclass of the target object.

Example MethodInterceptor intercept() Method

```
import org.springframework.cglib.proxy.MethodProxy;
import java.lang.reflect.Method;
...

@Override
public Object intercept(Object proxy, Method method, Object[] args,
                       MethodProxy methodProxy) throws Throwable {
    //do work ...
    log.info("invoke calling: {}({})", method.getName(), args);

    Object result = methodProxy.invokeSuper(proxy, args); ①
```

```

    //do work ...
    log.info("invoke {} returned: {}", method.getName(), result);

    //return result
    return result;
}

```

① invoking target object implementation in base class

246.4. Calling CGLIB Proxied Object

The net result is that we are still able to reach the target object's method and also have the additional capability implemented around the call of that method.

Example CGLIB Proxied Object Call

```

ChairDTO createdChair = chairsServiceProxy.createItem(
    ChairDTO.chairBuilder().name("Recliner").build());
log.info("created chair: {}", createdChair);

```

Example CGLIB Proxied Object Call Output

```

invoke calling: createItem([ChairDTO(super=ItemDTO(id=0, name=Recliner))])
invoke createItem returned: ChairDTO(super=ItemDTO(id=1, name=Recliner))
created chair: ChairDTO(super=ItemDTO(id=1, name=Recliner))

```

246.5. Dynamic Object CGLIB Proxy

The CGLIB example above formed a proxy around the base class. That is a unique feature to CGLIB because it can proxy no interface classes. If you remember, Dynamic JDK Proxies can only proxy interfaces and must be handed the instance to proxy at runtime. The proxied object is constructed separate from the Dynamic JDK Proxy and then wrapped by the [InvocationHandler](#).

Review: Dynamic JDK Interface Proxy Wrapping Existing Object

```

ItemsService<GrillDTO> grillService = new GrillsServiceImpl(); ①

ItemsService<GrillDTO> grillServiceProxy = (ItemsService<GrillDTO>)
    Proxy.newProxyInstance(
        grillService.getClass().getClassLoader(),
        new Class[]{ItemsService.class},
        new MyInvocationHandler(grillService) ②
    );

```

① the proxied object

② Dynamic JDK Proxy configured to proxy existing object

The same thing is true about CGLIB. We can design a [MethodInterceptor](#) that accepts an existing

instance and ignores the base class. In this use, the no-interface base class (from a pure Java perspective) is being used as strictly an interface (from an integration perspective). This is helpful when we need to dynamically proxy an object that comes in the door.

CGLIB Proxy Wrapping Existing Object

```
ChairsServiceImpl chairsServiceToProxy = ...  
  
Enhancer enhancer = new Enhancer();  
enhancer.setSuperclass(ChairsServiceImpl.class);  
enhancer.setCallback(new MyMethodInterceptor( chairsServiceToProxy )); ①  
ChairsServiceImpl chairsServiceProxy = (ChairsServiceImpl)enhancer.create();
```

① we are free to design the **MethodInterceptor** to communicate with whatever we need

[1] "Introduction to cglib", Baeldung, Aug 2019

Chapter 247. AOP Proxy Factory

Spring AOP offers a programmatic way to set up proxies using a [ProxyFactory](#) and allow Spring to determine which proxy technique to use. This becomes an available option once we add the [spring-boot-starter-aop](#) dependency.

Spring AOP Maven Dependency

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```

The following snippet shows a basic use case where the code locates a service and makes a call to that service. No proxy is used during this example. I just wanted to set the scene to get started.

Basic Use Case - No Proxy

```
MowerDTO mower1 = MowerDTO.mowerBuilder().name("John Deer").build();
ItemsService<MowerDTO> mowerService = ...

mowerService.createItem(mower1);
```

With AOP, we can write advice by implementing [numerous interfaces](#) with methods that are geared towards the lifecycle of a method call (e.g., before calling, after success). The following snippet shows a single advice class implementing a few advice methods. We will cover the specific advice methods more in the later declarative Spring AOP section.

Example Before and After Return Advice

```
import org.springframework.aop.AfterReturningAdvice;
import org.springframework.aop.MethodBeforeAdvice;
import java.lang.reflect.Method;
...
public class SampleAdvice1 implements MethodBeforeAdvice, AfterReturningAdvice {
    @Override
    public void before(Method method, Object[] args, Object target) throws Throwable {
        log.info("before: {}.{}({})", target, method, args);
    }

    @Override
    public void afterReturning(Object returnValue, Method method, Object[] args,
Object target) throws Throwable {
        log.info("after: {}.{}({}) = {}", target, method, args, returnValue);
    }
}
```

Using a Spring AOP [ProxyFactory](#), we can programmatically assemble an AOP Proxy with the advice

that will either use JDK Dynamic Proxy or CGLIB.

```
import org.springframework.aop.framework.ProxyFactory;  
  
...  
MowerDTO mower2 = MowerDTO.mowerBuilder().name("Husqvarna").build();  
ItemsService<MowerDTO> mowerService = ... ①  
  
ProxyFactory proxyFactory = new ProxyFactory(mowerService); ②  
proxyFactory.addAdvice(new SampleAdvice1()); ③  
  
ItemsService<MowerDTO> proxiedMowerService =  
    (ItemsService<MowerDTO>) proxyFactory.getProxy(); ④  
proxiedMowerService.createItem(mower2); ⑤  
  
log.info("proxy class={}", proxiedMowerService.getClass());
```

- ① this service will be the target of the proxy
- ② use **ProxyFactory** to assemble the proxy
- ③ assign one or more advice to the target
- ④ obtain reference to the proxy
- ⑤ invoke the target via the proxy

Spring AOP will determine the best approach to implement the proxy and produce one that is likely based on either the JDK Dynamic Proxy or CGLIB. The following snippet shows a portion of the **proxyFactory.getProxy()** call where the physical proxy is constructed.

Construction of Either Dynamic Proxy or CGLIB Proxy

```
package org.springframework.aop.framework;  
...  
public AopProxy createAopProxy(AdvisedSupport config) throws AopConfigException {  
    if (config.isOptimize() || config.isProxyTargetClass() ||  
hasNoUserSuppliedProxyInterfaces(config)) {  
    ...  
        return new ObjenesisCglibAopProxy(config);  
    }  
    else {  
        return new JdkDynamicAopProxy(config);  
    }  
}
```

The following snippet shows the (before and afterReturn) output from the example target call using the proxy created by Spring AOP **ProxyFactory** and the name of the proxy class. From the output, we can determine that a JDK Dynamic Proxy was used to implement the proxy.

Before and AfterReturn Advice Output During Call to Target

```
SampleAdvice1#before:23 before:
```

```

info.ejava.examples.svc.aop.items.services.MowersServiceImpl@509c0153.public abstract
info.ejava.examples.svc.aop.items.dto.ItemDTO
info.ejava.examples.svc.aop.items.services.ItemsService.createItem(info.ejava.examples
.svc.aop.items.dto.ItemDTO)([MowerDTO(super=ItemDTO(id=0, name=Husqvarna))])①

SampleAdvice1#afterReturning:28 after:
info.ejava.examples.svc.aop.items.services.MowersServiceImpl@509c0153.public abstract
info.ejava.examples.svc.aop.items.dto.ItemDTO
info.ejava.examples.svc.aop.items.services.ItemsService.createItem(info.ejava.examples
.svc.aop.items.dto.ItemDTO)([MowerDTO(super=ItemDTO(id=0, name=Husqvarna))] =
MowerDTO(super=ItemDTO(id=2, name=Husqvarna))②

proxy class=class jdk.proxy3.$Proxy94 ③

```

① output of before advice

② output of after advice

③ output with name of proxy class

When you need to programmatically assemble proxies—Reflection, Dynamic Proxies, and CGLIB provide a wide set of tools to accomplish this and AOP [ProxyFactory](#) can provide a high level abstraction above them all. In the following sections, we will look at how we can automate this and have a little insight into how automation is achieving its job.

Chapter 248. Interpose

OK—all that dynamic method calling was interesting, but what sets all that up? Why do we see proxies sometimes and not other times in our Spring application? We will get to the setup in a moment, but let's first address when we can expect this type of behavior magically setup for us and not. What occurs automatically is primarily a matter of "interpose". Interpose is a term used when we have a chance to insert a proxy in between the caller and target object. The following diagram depicts three scenarios: buddy methods of the same class, calling method of manually instantiated class, and calling method of injected object.

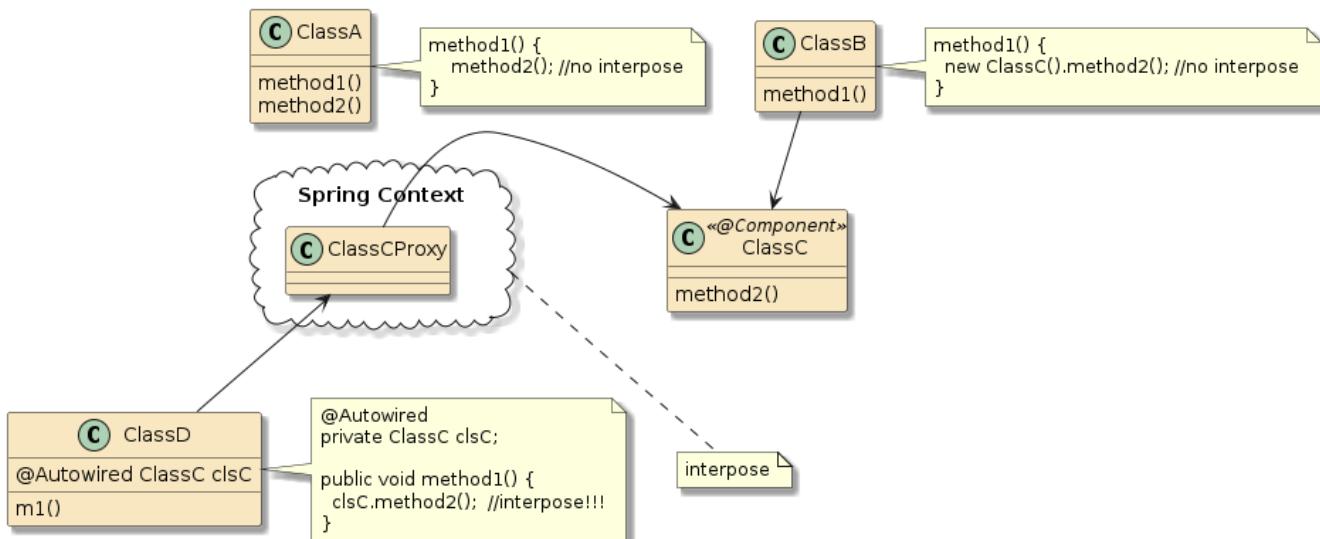


Figure 115. Interpose is only Automatic for Injected Components

- Buddy Method:** For the `ClassA` with `m1()` and `m2()` in the same class, Spring will normally not attempt to interpose a proxy in between those two methods (e.g., `@PreAuthorize`, `@Cacheable`). It is a straight Java call between methods of a class. That means no matter what annotations and constraints we define for `m2()` they will not be honored unless they are also on `m1()`. There is at least one exception for buddy methods, for `@Configuration(proxyBeanMethods=true)`—where a CGLIB proxy class will intercept calls between `@Bean` methods to prevent direct buddy method calls from instantiating independent POJO instances per call (i.e., not singleton components).
- Manually Instantiated:** For `ClassB` where `m2()` has been moved to a separate class but manually instantiated—no interpose takes place. This is a straight Java call between methods of two different classes. That also means that no matter what annotations are defined for `m2()`, they will not be honored unless they are also in place on `m1()`. It does not matter that `ClassC` is annotated as a `@Component` since `ClassB.m1()` manually instantiated it versus obtaining it from the Spring Context.
- Injected:** For `ClassD`, an instance of `ClassC` is injected. That means that the injected object has a chance to be a proxy class (either JDK Dynamic Proxy or CGLIB Proxy) to enforce the constraints defined on `ClassC.m2()`.

Keep this in mind as you work with various Spring configurations and review the following sections.

Chapter 249. Spring AOP

Spring Aspect Oriented Programming (AOP) provides a framework where we can define cross-cutting behavior to injected `@Components` using one or more of the available proxy capabilities behind the scenes. Spring AOP Proxy uses JDK Dynamic Proxy to proxy beans with interfaces and CGLIB to proxy bean classes lacking interfaces.

Spring AOP is a very capable but scaled back and simplified implementation of [AspectJ](#). All the capabilities of AspectJ are allowed within Spring. However, the features integrated into Spring AOP itself are limited to method proxies formed at runtime. The compile-time byte manipulation offered by AspectJ is not part of Spring AOP.

249.1. AOP Definitions

The following represent some core definitions to AOP. Advice, AOP proxy, target object and (conceptually) the join point should look familiar to you. The biggest new concept here is the pointcut predicate used to locate the join point and how that is all modularized through a concept called "aspect".

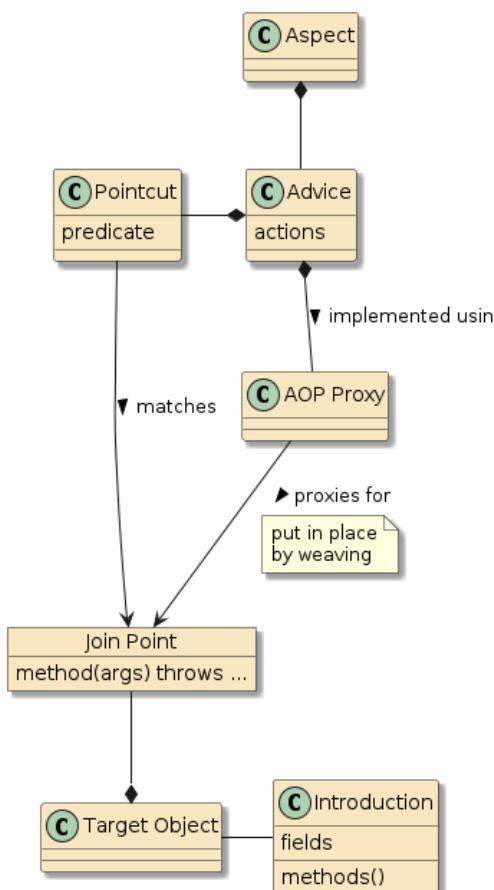


Figure 116. AOP Key Terms

Join Point is a point in the program, (e.g., calling a method or throwing exception) in which we want to inject some code. For Spring AOP — this is always an event related to a method. AspectJ offers more types of join points.

Pointcut is a predicate rule that matches against a join point (i.e., a method begin, success, exception, or finally) and associates advice (i.e., more code) to execute at that point in the program. Spring uses the AspectJ pointcut language.

Advice is an action to be taken at a join point. This can be before, after (success, exception, or always), or around a method call. Advice chains are formed much the same as Filter chains of the web tier.

AOP proxy is an object created by AOP framework to implement advice against join points that match the pointcut predicate rule.

Aspect is a modularization of a concern that cuts across multiple classes/methods (e.g., timing measurement, security auditing, transaction boundaries). An aspect is made up of one or more advice action(s) with an assigned pointcut predicate.

Target object is an object being advised by one or more aspects. Spring uses proxies to implement advised (target) objects.

Introduction is declaring additional methods or fields on behalf of a type for an advised object, allowing us to add an additional interface and implementation.

Weaving is the linking aspects to objects. Spring AOP does this at runtime. AspectJ offers compile-time capabilities.

249.2. Enabling Spring AOP

To use Spring AOP, we must first add a dependency on `spring-boot-starter-aop`. That adds a dependency on `spring-aop` and `aspectj-weaver`.

Spring AOP Maven Dependency

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```

We enable Spring AOP within our Spring Boot application by adding the `@EnableAspectJProxy` annotation to a `@Configuration` class or to the `@SpringBootApplication` class.

Enabling Spring AOP using Annotation

```
import org.springframework.context.annotation.EnableAspectJAutoProxy;
...
@Configuration
@EnableAspectJAutoProxy ①
public class ...
```

① add `@EnableAspectJAutoProxy` to a configuration class to enable dynamic AOP behavior in application

249.3. Aspect Class

Starting at the top—we have the `Aspect` class. This is a special `@Component` that defines the pointcut predicates to match and advice (before, after success, after throws, after finally, and around) to execute for join points.

Example Aspect Class

```
...
import org.aspectj.lang.annotation.Aspect;

@Component ①
@Aspect ②
public class ItemsAspect {
    //pointcuts
    //advice
```

```
}
```

- ① annotated `@Component` to be processed by the application context
- ② annotated as `@Aspect` to have pointcuts and advice inspected

249.4. Pointcut

In Spring AOP—a pointcut is a predicate rule that identifies the method join points to match against for Spring beans (only). To help reduce complexity of definition, when using annotations—pointcut predicate rules are expressed in two parts:

- pointcut expression that determines exactly which method executions we are interested in
- Java method signature—with name and parameters—to reference it

The signature is a method that returns void. The method name and parameters will be usable in later advice declarations. Although the abstract example below does not show any parameters, they will become quite useful when we begin injecting typed parameters.

Example Pointcut

```
import org.aspectj.lang.annotation.Pointcut;  
...  
@Pointcut(/* pointcut expression*/) ①  
public void serviceMethod(/* pointcut parameters */) {} //pointcut signature ②
```

- ① pointcut expression defines predicate matching rule(s)
- ② pointcut Java signature defines a name and parameter types for the pointcut expression

249.5. Pointcut Expression

The [Spring AOP](#) pointcut expressions use the [AspectJ](#) pointcut language. Supporting the following designators

• execution	match method execution join points
• within	match methods below a package or type
• @within	match methods of a type that has been annotated with a given annotation
• this	match the proxy for a given type—useful when injecting typed advice arguments when the proxy (not the target) implements a specific type. Proxies can implement more than just the target's interface.
• target	match the proxy's target for a given type—useful when injecting typed advice arguments when the target implements a specific type
• @target	match methods of a type that has been annotated with specific annotation
• @annotation	match methods that have been annotated with a given annotation
• args	match methods that accept arguments matching this criteria

• @args	match methods that accept arguments annotated with a given annotation
• bean	Spring AOP extension to match Spring bean(s) based on a name or wildcard name expression

Don't use pointcut contextual designators for matching



[Spring AOP Documentation](#) recommends we use `within` and/or `execution` as our first choice of performant predicate matching and add contextual designators (`args`, `@annotation`, `this`, `target`, etc.) when needed for additional work versus using contextual designators alone for matching.

249.6. Example Pointcut Definition

The following example will match against any method in the service's package, taking any number of arguments and returning any return type.

Example execution Pointcut

```
//execution(<return type> <package>.<class>.<method>(params)
@Pointcut("execution(* info.ejava.examples.svc.aop.items.services.*.*(..))")
//expression
public void serviceMethod() {} //signature
```

249.7. Combining Pointcut Expressions

We can combine pointcut definitions into compound definitions by referencing them and joining with a boolean ("`&&`" or "`||`") expression. The example below adds an additional condition to `serviceMethod()` that restricts matches to methods accepting a single parameter of type `GrillDTO`.

Example Combining Pointcut Expressions

```
@Pointcut("args(info.ejava.examples.svc.aop.items.dto.GrillDTO)") //expression
public void grillArgs() {} //signature

@Pointcut("serviceMethod() && grillArgs()") //expression
public void serviceMethodWithGrillArgs() {} //signature
```

Use as Example of Combining Two Pointcut Expressions



Use this example as an example of combining two pointcut expressions. Forming a matching expression using the contextual `args()` feature works, but is in violation of Spring AOP Documentation performance recommendations. Use contextual `args()` feature to identify portions of the call to pass into the advice. That will be shown soon.

249.8. Advice

The code that will act on the join point is specified in a method of the `@Aspect` class and annotated with one of the advice annotations. The following is an example of advice that executes before a join point.

Example Advice

```
...
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Component
@Aspect
@Slf4j
public class ItemsAspect {
    ...
    @Before("serviceMethodWithGrillArgs()")
    public void beforeGrillServiceMethod() {
        log.info("beforeGrillServiceMethod");
    }
}
```

The following table contains a list of the available advice types:

Table 17. Available Advice Types

<code>@Before</code>	runs prior to calling join point
<code>@AfterReturning</code>	runs after successful return from join point
<code>@AfterThrowing</code>	runs after exception from join point
<code>@After</code>	runs after join point no matter — i.e., finally
<code>@Around</code>	runs around join point. Advice must call join point and return result.

An example of each is towards the end of these lecture notes. For now, let's go into detail on some of the things we have covered.

Chapter 250. Pointcut Expression Examples

Pointcut expressions can be very expressive and can take some time to fully understand. The following examples should provide a head start in understanding the purpose of each and how they can be used. Other examples are available in the [Spring AOP](#) page.

250.1. execution Pointcut Expression

The execution expression allows for the definition of several pattern elements that can identify the point of a method call. The full format is as follows. ^[1]

execution Pointcut Expression Elements

```
execution(modifiers-pattern? ret-type-pattern declaring-type-pattern?name-
pattern(param-pattern) throws-pattern?)
```

However, only the return type, name, and parameter definitions are required.

Required execution Patterns

```
execution(ret-type-pattern name-pattern(param-pattern))
```

The specific patterns include:

- **modifiers-pattern** - OPTIONAL access definition (e.g., public, protected)
- **ret-type-pattern** - MANDATORY type pattern for return type

Example Return Type Patterns

```
execution(info.ejava.examples.svc.aop.items.dto.GrillDTO *(...)) ①
execution(*..GrillDTO *(...)) ②
```

① matches methods that return an explicit type

② matches methods that return `GrillDTO` type from any package

- **declaring-type-pattern** - OPTIONAL type pattern for package and class

Example Declaring Type (package and class) Pattern

```
execution(* info.ejava.examples.svc.aop.items.services.GrillsServiceImpl.*(...)) ①
execution(* *..GrillsServiceImpl.*(...)) ②
execution(* info.ejava.examples.svc..Grills*.*(...)) ③
```

① matches methods within an explicit class

② matches methods within a `GrillsServiceImpl` class from any package

③ matches methods from any class below `...svc` and start with letters `Grills`

- **name-pattern** - MANDATORY pattern for method name

Example Name (method) Pattern

```
execution(* createItem(..)) ①
execution(* ..GrillsServiceImpl.createItem(..)) ②
execution(* create*(..)) ③
```

- ① matches any method called `createItem` of any class of any package
- ② matches any method called `createItem` within class `GrillsServiceImpl` of any package
- ③ matches any method of any class of any package that starts with the letters `create`

- **param-pattern** - MANDATORY pattern to match method arguments. `()` will match a method with no arguments. `(*)` will match a method with a single parameter. `(T, *)` will match a method with two parameters with the first parameter of type `T`. `(..)` will match a method with zero (0) or more parameters

Example noargs () Pattern

```
execution(void
info.ejava.examples.svc.aop.items.services.GrillsServiceImpl.deleteItems())①
execution(* ..GrillsServiceImpl.*()) ②
execution(* ..GrillsServiceImpl.delete*()) ③
```

- ① matches an explicit method that takes no arguments
- ② matches any method within a `GrillsServiceImpl` class of any package and takes no arguments
- ③ matches any method from the `GrillsServiceImpl` class of any package, taking no arguments, and the method name starts with `delete`

Example Single Argument Patterns

```
execution(*
info.ejava.examples.svc.aop.items.services.GrillsServiceImpl.createItem(*))①
execution(* createItem(info.ejava.examples.svc.aop.items.dto.GrillDTO)) ②
execution(* *(..GrillDTO)) ③
```

- ① matches an explicit method that accepts any single argument
- ② matches any method called `createItem` that accepts a single parameter of a specific type
- ③ matches any method that accepts a single parameter of `GrillDTO` from any package

Example Multiple Argument Patterns

```
execution(*
info.ejava.examples.svc.aop.items.services.GrillsServiceImpl.updateItem(*, *))①
execution(* updateItem(int, *)) ②
execution(* updateItem(int, *..GrillDTO)) ③
```

- ① matches an explicit method that accepts two arguments of any type
- ② matches any method called `updateItem` that accepts two arguments of type `int` and any second type
- ③ matches any method called `updateItem` that accepts two arguments of type `int` and `GrillDTO` from any package

250.2. within Pointcut Expression

The within pointcut expression is similar to supplying an `execution` expression with just the declaring type pattern specified.

Example within Expressions

```
within(info.ejava.examples.svc.aop.items..*) ①
within(*..ItemsService+) ②
within(*..BedsServiceImpl) ③
```

- ① match all methods in package `info.ejava.examples.svc.aop.items` and its subpackages
- ② match all methods in classes that implement `ItemsService` interface
- ③ match all methods in `BedsServiceImpl` class

250.3. target and this Pointcut Expressions

The `target` and `this` pointcut designators are very close in concept to `within`. Like JDK Dynamic Proxy and CGLIB, AOP proxies can implement more interfaces (via Introductions) than the target being proxied. `target` refers to the object being proxied. `this` refers to the proxy. These are considered "contextual" designators and are primarily placed in the predicate to pull out members of the call for injection.

Example target and this Patterns

```
target(info.ejava.examples.svc.aop.items.services.BedsServiceImpl) ①
this(info.ejava.examples.svc.aop.items.services.BedsServiceImpl) ②
```

- ① matches methods of target object — object being proxied — is of type
- ② matches methods of proxy object — object implementing proxy — is of type

Example @target and @annotation Patterns

```
@target(org.springframework.stereotype.Service) ①
@annotation(org.springframework.core.annotation.Order) ②
```

- ① matches all methods in class annotated with `@Service` and implemented by target object
- ② matches all methods having annotation `@Order`

[1] "Spring AOP execution Examples", Spring AOP

Chapter 251. Advice Parameters

Our advice methods can accept two types of parameters:

- typed using context designators
- dynamic using [JoinPoint](#)

Context designators like `args`, `@annotation`, `target`, and `this` allow us to assign a logical name to a specific part of a method call so that can be injected into our advice method.

Dynamic injection involves a single `JointPoint` object that can answer the contextual details of the call.



Do not use context designators alone as predicates to locate join points

The Spring AOP documentation recommends using `within` and `execution` designators to identify a pointcut and contextual designators like `args` to bind aspects of the call to input parameters. That guidance is not fully followed in the following context examples. We easily could have made the non-contextual designators more explicit.

251.1. Typed Advice Parameters

We can use the `args` expression in the pointcut to identify criteria for parameters to the method and to specifically access one or more of them.

The left side of the following pointcut expression matches on all executions of methods called `createGrill()` taking any number of arguments. The right side of the pointcut expression matches on methods with a single argument. When we match that with the `createGrill` signature—the single argument must be of the type `GrillDTO`.

Example Single, Typed Argument

```
@Pointcut("execution(* createItem(..)) && args(grillDTO)") ① ②
public void createGrill(GrillDTO grillDTO) {} ③

@Before("createGrill(grill)") ④
public void beforeCreateGrillAdvice(GrillDTO grill) { ⑤
    log.info("beforeCreateGrillAdvice: {}", grill);
}
```

① left hand side of pointcut expression matches execution of `createItem` methods with any parameters

② right hand side of pointcut expression matches methods with a single argument and maps that to name `grillDTO`

③ pointcut signature maps `grillDTO` to a Java type—the names within the pointcut must match

④ advice expression references `createGrill` pointcut and maps first parameter to name `grill`

- ⑤ advice method signature maps name `grill` to a Java type—the names within the advice must match but do not need to match the names of the pointcut

The following is logged before the `createGrill` method is called.

Example Single, Typed Argument Output

```
beforeCreateGrillAdvice: GrillDTO(super=ItemDTO(id=0, name=weber))
```

251.2. Multiple, Typed Advice Parameters

We can use the `args` designator to specify multiple arguments as well. The right hand side of the pointcut expression matches methods that accept two parameters. The pointcut method signature maps these to parameters to Java types. The example advice references the pointcut but happens to use different parameter names. The names used match the parameters used in the advice method signature.

Example Multiple, Typed Arguments

```
@Pointcut("execution(* updateItem(..)) && args(grillId, updatedGrill)")  
public void updateGrill(int grillId, GrillDTO updatedGrill) {}  
  
@Before("updateGrill(id, grill)")  
public void beforeUpdateGrillAdvice(int id, GrillDTO grill) {  
    log.info("beforeUpdateGrillAdvice: {}, {}", id, grill);  
}
```

The following is logged before the `updateGrill` method is called.

Example Multiple, Typed Arguments Output

```
beforeUpdateGrillAdvice: 1, GrillDTO(super=ItemDTO(id=0, name=green egg))
```

251.3. Annotation Parameters

We can target annotated classes and methods and make the value of the annotation available to the advice using the pointcut signature mapping. In the example below, we want to match on all methods below the `items` package that have an `@Order` annotation and pass that annotation as a parameter to the advice.

Example @Annotation Parameter

```
import org.springframework.core.annotation.Order;  
...  
@Pointcut("@annotation(order)") ①  
public void orderAnnotationValue(Order order) {} ②  
  
@Before("within(info.ejava.examples.svc.items..*) && orderAnnotationValue(order)")
```

```

public void beforeOrderAnnotation(Order order) { ③
    log.info("before@OrderAnnotation: order={}, order.value()); ④
}

```

- ① we are targeting methods with an annotation and mapping that to the name `order`
- ② the name `order` is being mapped to the type `org.springframework.core.annotation.Order`
- ③ the `@Order` annotation instance is being passed into advice
- ④ the value for the `@Order` annotation can be accessed

I have annotated one of the candidate methods with the `@Order` annotation and assigned a value of `100`.

Example @Annotation Parameter Target Method

```

import org.springframework.core.annotation.Order;
...
@Service
public class BedsServiceImpl extends ItemsServiceImpl<BedDTO> {
    @Override
    @Order(100)
    public BedDTO createItem(BedDTO item) {

```

In the output below—we see that the annotation was passed into the advice and provided with the value `100`.

Example @Annotation Parameter Output

```
before@OrderAnnotation: order=100
```

Annotations can pass contextual values to advice



Think how a feature like this—where an annotation on a method with attribute values—can be of use with security role annotations (`@PreAuthorize(hasRole('ADMIN'))`).

251.4. Target and Proxy Parameters

We can map the target and proxy references into the advice method using the `target()` and `this()` designators. In the example below, the `target` name is mapped to the `ItemsService<BedDTO>` interface and the `proxy` name is mapped to a vanilla `java.lang.Object`. The `target` type mapping constrains this call to the `BedsServiceImpl` object being proxied.

Example target and this Parameters

```

@Before("target(target) && this(proxy)")
public void beforeTarget(ItemsService<BedDTO> target, Object proxy) {
    log.info("beforeTarget: target={}, proxy={}", target.getClass(), proxy.getClass());
}

```

```
}
```

The advice prints the name of each class. The output below shows that the target is of the target implementation type and the proxy is of a CGLIB proxy type (i.e., it is the proxy to the target).

Example target and this Parameters Result

```
beforeTarget:  
target=class info.ejava.examples.svc.aop.items.services.BedsServiceImpl,  
proxy=class  
info.ejava.examples.svc.aop.items.services.BedsServiceImpl$$EnhancerBySpringCGLIB$$a38  
982b5
```

251.5. Dynamic Parameters

If we have generic pointcuts and do not know ahead of time which parameters we will get and in what order, we can inject a [JoinPoint](#) parameter as the first argument to the advice. This object has many methods that provide dynamic access to the context of the method—including parameters. The example below logs the classname, method, and array of parameters in the call.

Example JointPoint Injection

```
@Before("execution(* *..Grills*.*(..))")  
public void beforeGrillsMethodsUnknown(JoinPoint jp) {  
    log.info("beforeGrillsMethodsUnknown: {}.{}. {}, {}",  
            jp.getTarget().getClass().getSimpleName(),  
            jp.getSignature().getName(),  
            jp.getArgs());  
}
```

251.6. Dynamic Parameters Output

The following output shows two sets of calls: `createItem` and `updateItem`. Each was intercepted at the controller and service level.

Example JointPoint Injection Output

```
beforeGrillsMethodsUnknown: GrillsController.createItem,  
                                [GrillDTO(super=ItemDTO(id=0, name=weber))]  
beforeGrillsMethodsUnknown: GrillsServiceImpl.createItem,  
                                [GrillDTO(super=ItemDTO(id=0, name=weber))]  
beforeGrillsMethodsUnknown: GrillsController.updateItem,  
                                [1, GrillDTO(super=ItemDTO(id=0, name=green egg))]  
beforeGrillsMethodsUnknown: GrillsServiceImpl.updateItem,  
                                [1, GrillDTO(super=ItemDTO(id=0, name=green egg))]
```

Chapter 252. Advice Types

We have five advice types:

- @Before
- @AfterReturning
- @AfterThrowing
- @After
- @Around

For the first four—using `JoinPoint` is optional. The last type (`@Around`) is required to inject `ProceedingJoinPoint`—a subclass of `JoinPoint`—to delegate to the target and handle the result. Let's take a look at each to have a complete set of examples.

To demonstrate, I am going to define advice of each type that will use the same pointcut below.

Example Pointcut to Demonstrate Advice Types

```
@Pointcut("execution(* *..MowersServiceImpl.updateItem(*,*)) && args(id,mowerUpdate)"  
①  
public void mowerUpdate(int id, MowerDTO mowerUpdate) {} ②
```

① matches all `updateItem` methods calls in the `MowersServiceImpl` class taking two arguments

② arguments will be mapped to type `int` and `MowerDTO`

There will be two matching update calls:

1. the first will be successful and return a result
2. the second will throw an example `NotFoundException`

252.1. @Before

The Before advice will be called prior to invoking the join point method. It has access to the input parameters and can change the contents of them. This advice does not have access to the result.

Example @Before Advice

```
@Before("mowerUpdate(id, mowerUpdate)")  
public void beforeMowerUpdate(JoinPoint jp, int id, MowerDTO mowerUpdate) {  
    log.info("beforeMowerUpdate: {}, {}", id, mowerUpdate);  
}
```

The before advice only has access to the input parameters prior to making the call. It can modify the parameters, but not swap them around. It has no insight into what the result will be.

@Before Advice Example for Successful Call

```
beforeMowerUpdate: 1, MowerDTO(super=ItemDTO(id=0, name=bush hog))
```

Since the before advice is called prior to the join point, it is oblivious that this call ended in an exception.

@Before Advice Example for Call Throwing Exception

```
beforeMowerUpdate: 2, MowerDTO(super=ItemDTO(id=0, name=john deer))
```

252.2. @AfterReturning

After returning, advice will get called when a join point successfully returns without throwing an exception. We have access to the result through an annotation field and can map that to an input parameter.

Example @AfterReturning Advice

```
@AfterReturning(value = "mowerUpdate(id, mowerUpdate)",  
    returning = "result")  
public void afterReturningMowerUpdate(JoinPoint jp, int id, MowerDTO mowerUpdate,  
MowerDTO result) {  
    log.info("afterReturningMowerUpdate: {}, {} => {}", id, mowerUpdate, result);  
}
```

The **@AfterReturning** advice is called only after the successful call and not the exception case. We have access to the input parameters and the result. The result can be changed before returning to the caller. However, the input parameters have already been processed.

@AfterReturning Advice Example for Successful Call

```
afterReturningMowerUpdate: 1, MowerDTO(super=ItemDTO(id=1, name=bush hog))  
=> MowerDTO(super=ItemDTO(id=1, name=bush hog))
```

252.3. @AfterThrowing

The **@AfterThrowing** advice is called only when an exception is thrown. Like the successful sibling, we can map the resulting exception to an input variable to make it accessible to the advice.

Example @AfterThrowing Advice

```
@AfterThrowing(value = "mowerUpdate(id, mowerUpdate)", throwing = "ex")  
public void afterThrowingMowerUpdate(JoinPoint jp, int id, MowerDTO mowerUpdate,  
ClientErrorException.NotFoundException ex) { ①  
    log.info("afterThrowingMowerUpdate: {}, {} => {}", id, mowerUpdate, ex.toString());
```

```
}
```

- ① advice will only be called if `NotFoundException` is thrown — otherwise skipped

The `@AfterThrowing` advice has access to the input parameters and the exception. The exception will still be thrown after the advice is complete. I am not aware of any ability to squelch the exception and return a non-exception here. Look to `@Around` to give you that capability at a minimum.

@AfterThrowing Advice Example for Call Throwing Exception

```
afterThrowingMowerUpdate: 2, MowerDTO(super=ItemDTO(id=0, name=john deer))
    => info.ejava.examples.common.exceptions.ClientErrorException$NotFoundException:
item[2] not found
```

252.4. `@After`

`@After` is called after a successful return or exception thrown. It represents logic that would commonly appear in a `finally` block to close out resources.

Example @After Advice

```
@After("mowerUpdate(id, mowerUpdate)")
public void afterMowerUpdate(JoinPoint jp, int id, MowerDTO mowerUpdate) {
    log.info("afterReturningMowerUpdate: {}, {}", id, mowerUpdate);
}
```

The `@After` advice is always called once the joint point finishes executing. It does not provide an indication of whether the method completed or threw an exception.

@After Advice Example for Successful Call

```
afterReturningMowerUpdate: 1, MowerDTO(super=ItemDTO(id=1, name=bush hog))
```

@After Advice Example for Call Throwing Exception

```
afterReturningMowerUpdate: 2, MowerDTO(super=ItemDTO(id=0, name=john deer))
```

@After is not Informed of Success or Failure

i `@After` callbacks do not provide a sign of success/failure when they are called. Treat this like a finally clause.

252.5. `@Around`

`@Around` is the most capable advice but possibly the most expensive/complex to execute. It has full control over the input and return values and whether the call is made at all. The example below logs the various paths through the advice.

Example @Around Advice

```
@Around("mowerUpdate(id, mowerUpdate)")  
public Object aroundMowerUpdate(ProceedingJoinPoint pjp, int id, MowerDTO mowerUpdate)  
throws Throwable {  
    Object result = null;  
    try {  
        log.info("entering aroundMowerUpdate: {}, {}", id, mowerUpdate);  
        result = pjp.proceed(pjp.getArgs());  
        log.info("returning after successful aroundMowerUpdate: {}, {} => {}", id,  
mowerUpdate, result);  
        return result;  
    } catch (Throwable ex) {  
        log.info("returning after aroundMowerUpdate exception: {}, {} => {}", id,  
mowerUpdate, ex.toString());  
        throw ex;  
    } finally {  
        log.info("returning after aroundMowerUpdate: {}, {} => {}",  
                id, mowerUpdate, (result==null ? null :result.toString()));  
    }  
}
```

The **@Around** advice example will log activity before calling the join point, after successful return from join point, and finally after all advice completes.

@Around Advice Example for Successful Call

```
entering aroundMowerUpdate: 1, MowerDTO(super=ItemDTO(id=0, name=bush hog))  
returning after successful aroundMowerUpdate: 1, MowerDTO(super=ItemDTO(id=1,  
name=bush hog))  
=> MowerDTO(super=ItemDTO(id=1, name=bush hog))  
returning after aroundMowerUpdate: 1, MowerDTO(super=ItemDTO(id=1, name=bush hog))  
=> MowerDTO(super=ItemDTO(id=1, name=bush hog))
```

The **@Around** advice example will log activity before calling the join point, after an exception from the join point, and finally after all advice completes.

@Around Advice Example for Call Throwing Exception

```
entering aroundMowerUpdate: 2, MowerDTO(super=ItemDTO(id=0, name=john deer))  
returning after aroundMowerUpdate exception: 2, MowerDTO(super=ItemDTO(id=0, name=john  
deer))  
=> info.ejava.examples.common.exceptions.ClientErrorException$NotFoundException:  
item[2] not found  
returning after aroundMowerUpdate: 2, MowerDTO(super=ItemDTO(id=0, name=john deer))  
=> info.ejava.examples.common.exceptions.ClientErrorException$NotFoundException:  
item[2] not found
```

Chapter 253. Introductions

JDK Dynamic Proxy and CGLIB provide a means to define interfaces implemented by the proxy. It was the callback handler's job to deliver that method call to the correct location. It was easily within the callback handler's implementation ability to delegate to additional object state within the callback handler instance. The integration of these independent types is referred to as a [mixin](#).

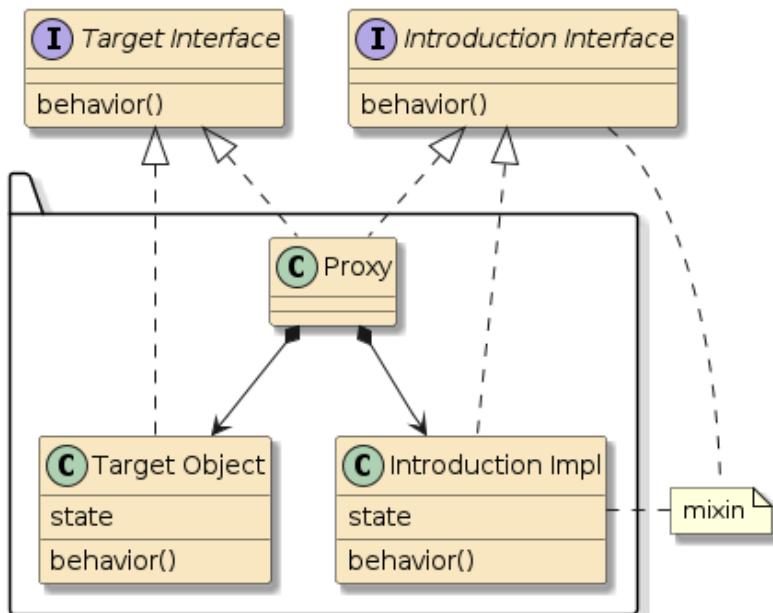


Figure 117. Introductions

Introductions enable you to implement a mixin. One can define a target object to implement additional interfaces and provide an implementation for the additional interface(s). This can be handled:

- using a declarative `@DeclareParents` construct for component targets in the Spring context
- with the aid of AOP advice and the `ProxyFactory` we discussed earlier for data targets.

253.1. Component Introductions

Introductions can be automatically applied to any component being processed by Spring via declarative AOP.

253.1.1. Component Introduction Declaration

The following snippet shows a `MyUsageIntroduction` interface that will be applied to component classes matching the supplied pattern. We are also assigning a required `MyUsageIntroductionImpl` implementation that will be the target of the `MyUsageIntroduction` interface calls.

Example `MyUsageIntroduction` Definition

```
@DeclareParents(value="info.ejava.examples.svc.aop.items.services.*", ①
    defaultImpl = MyUsageIntroductionImpl.class) ②
public static MyUsageIntroduction mixin; ③
```

① service class pattern to apply Introduction

② implementation for Introduction interface

③ Java interface type of the Introduction added to the target component—`mixin` name is not used

253.1.2. Introduction POJO Interface/Implementation

The interface and implementation are nothing special. The implementation is a standard POJO. One thing to note, however, is that the methods in this Introduction interface need to be unique relative to the target. Any duplication in method signatures will be routed to the target and not the Introduction.

For the example, we will implement some state and methods that will be assigned to each advised target component. The following snippets show the interface and implementation that will be used to track calls made to the target.

Example Component Introduction Interface Example Introduction Implementation

```
public interface  
MyUsageIntroduction {  
    List<String> getAllCalled();  
    void addCalled(String called);  
    void clear();  
}
```

The calls are being tracked as a simple collection of Strings. The client, having access to the service, can also have access to the Introduction methods and state.

```
public class MyUsageIntroductionImpl  
    implements MyUsageIntroduction  
{  
    private List<String> calls = new ArrayList<>();  
    @Override  
    public List<String> getAllCalled() {  
        return new ArrayList<>(calls);  
    }  
    @Override  
    public void addCalled(String called) {  
        calls.add(called);  
    }  
    @Override  
    public void clear() { calls.clear();  
    }  
}
```

Introduction and Target Methods must have Distinct/Unique Signatures



Introduction interface methods must have a signature distinct from the target or else the Introduction will be bypassed.

At this point—we are done enhancing the component with an Introduction interface and implementation. The diagram below shows most of the example all together.

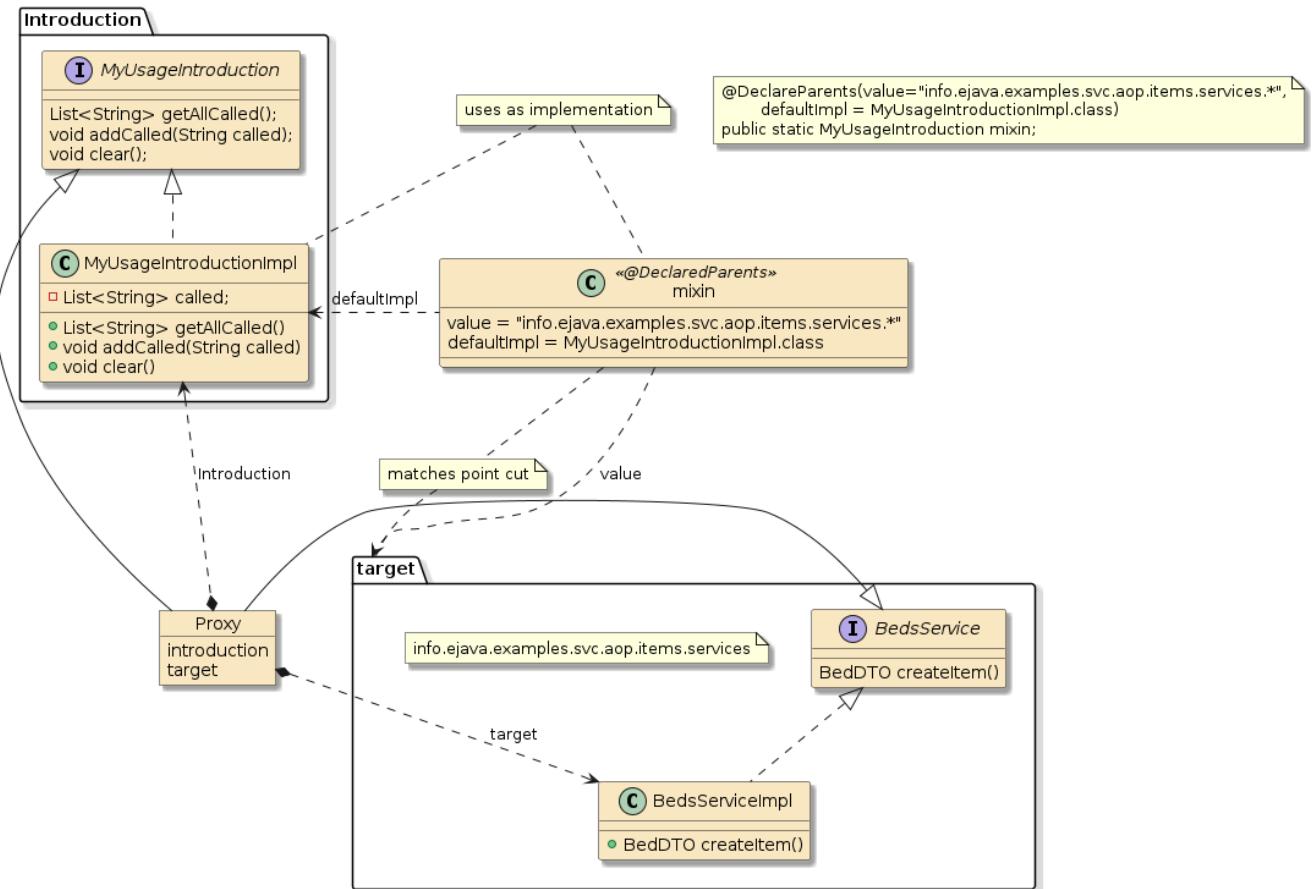


Figure 118. @DeclareParent Resulting Proxy

The remaining parts of the example have to do with using that additional "mixin" behavior via the proxy.

253.1.3. Component Introduction Example AOP Use

With the Introduction being applied to all component classes matching the pattern above, we are going to interact with the Introduction for all calls to `createItem()`. We will add code to identify the call being made and log that in the Introduction.

The following snippet shows a `@Pointcut` expression that will match all `createItem()` method calls within the same package defined to have the Introduction. The use of AOP here is independent of our proxied target. We can access the same methods from everywhere the proxy is injected—as you will see when we make calls shortly.

Pointcut for Example Introduction Trigger

```
@Pointcut("execution(* info.ejava.examples.svc.aop.items.services.*.createItem(..))")
public void anyCreateItem() {}
```

The advice associated with the pointcut, shown in the snippet below, will be called before any `createItem()` calls against a proxy implementing the `MyUsageIntroduction` interface. The call is injected with the proxy using the Introduction interface. The advice is oblivious to the actual proxy target type. This example client uses the `JoinPoint` object to identify the call being made to the proxy and places that into the Introduction list of calls.

Advice for Example Introduction Trigger

```
@Before("anyCreateItem() && this(usageIntro)") ①
public void recordCalled(JoinPoint jp, MyUsageIntroduction usageIntro) { ②
    Object arg = jp.getArgs()[0];
    usageIntro.addCalled(jp.getSignature().toString() + ":" + arg); ③
}
```

- ① apply advice to `createItem` calls to proxies implementing the `MyUsageIntroduction` interface
- ② call advice and inject a reference using Introduction type
- ③ use the Introduction to add information about the call being made

this() References the Proxy, *target()* References the target



The contextual `this` designator matches any proxy implementing the `MyUsageIntroduction`. The contextual `target` designator would match any target implementing the designated interface. In this case, we must match on the proxy.

253.1.4. Injected Component Example Introduction Use

With the AOP `@Before` advice in place, we can invoke Spring injected `bedsService` to trigger the extra Introduction behavior. The `bedsService` can be successfully cast to `MyUsageIntroduction` since this instance is a Spring proxy implementing the two (2) interfaces. Using a handle to the `MyUsageIntroduction` interface we can make use of the additional Introduction information stored with the `bedService` component.

Client Using Introduction from Injected Component

```
@Autowired
private ItemsService<BedDTO> bedsService; ①
...
BedDTO bed = BedDTO.bedBuilder().name("Single Bed").build();

BedDTO createdBed = bedsService.createItem(bed); ②

MyUsageIntroduction usage = (MyUsageIntroduction) bedsService; ③
String signature = String.format("BedDTO %s.createItem(BedDTO):%s",
    BedsServiceImpl.class.getName(), bed);

then( usage.getAllCalled() ).containsExactly(signature); ④
```

- ① Introduction was applied to component by Spring
- ② call to `createItem` will trigger `@Before` advice
- ③ component can be cast to Introduction interface
- ④ verification that Introduction was successfully applied to component

Using Introductions for Spring component targets allows us to not only add disparate advice behavior—but also retain state specific to each target with the target.

253.2. Data Introductions

Introductions can also be applied to data. Although it is not technically the same proxy mechanism used with Spring AOP, if you have any familiarity with Hibernate and "managed entities", you will be familiar with the concept of adding behavior and state on a per-target data object basis ([Hibernate transitioned from CGLIB to Byte Buddy](#)). We have to push some extra peddles to proxy data since data passed to or returned from component calls are not automatically subject to interpose.

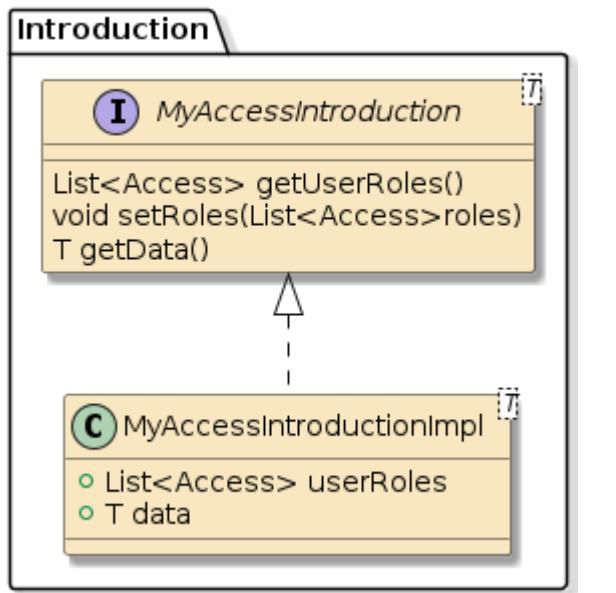
253.2.1. Example Data Introduction

The following snippet shows the core of an example Introduction interface and implementation that will be added to the target data objects. It defines a set of accesses the attributed user of the call has relative to the target data object. We wish to store this state with the target data object.

Table 18. Example Data Introduction Interface

```
public interface MyAccessIntroduction<T>
{
    enum Access { MEMBER, OWNER, ADMIN }
    List<Access> getUserRoles();
    void setUserRoles(List<Access>
roles);
    T getData(); ①
```

① Introduction Accesses will augment target



The implementation below shows the Introduction implementation being given direct access to the target data object (`T data`). This will be shown later in the example. Of note, since the `toString()` method duplicates the signature from the `target.toString()`, the Introduction's `toString()` will be bypassed for the target's by the proxy.

Example Data Introduction Implementation

```
@RequiredArgsConstructor
@Getter
@ToString ②
public class MyAccessIntroductionImpl<T> implements MyAccessIntroduction<T> {
    private final List<Access> userRoles = new ArrayList<>();
    private final T data; ①
    @Override
    public void setUserRoles(List<Access> roles) { ...
    ...
}
```

- ① optional direct access to target data use will be shown later
- ② contributes no value; duplicates `target.toString()`; will never be called via proxy

253.2.2. Intercepting Data Target

Unlike the managed components, Spring does not have an automated way to add Introductions to objects returned from methods. We will set up custom intercept using Spring AOP Advice and use manual calls to the `ProxyFactory` discussed earlier to build our proxy. The following snippet shows a `@Pointcut` pattern that will match all `getItem()` methods in our target component package. This is matching the service/method that will return the target data object.

Example Pointcut for Intercepting Data Target

```
@Pointcut("execution(* info.ejava.examples.svc.aop.items.services.*.getItem(..))")
public void getter() {}
```

The figure below shows the high level relationships and assembly. The AOP advice:

1. intercepts call to `BedService.createItem()`
2. builds a `ProxyFactory` to construct a proxy
3. assigns the target to be advised by the proxy
4. adds an Introduction interface
5. adds Introduction implementation as advice

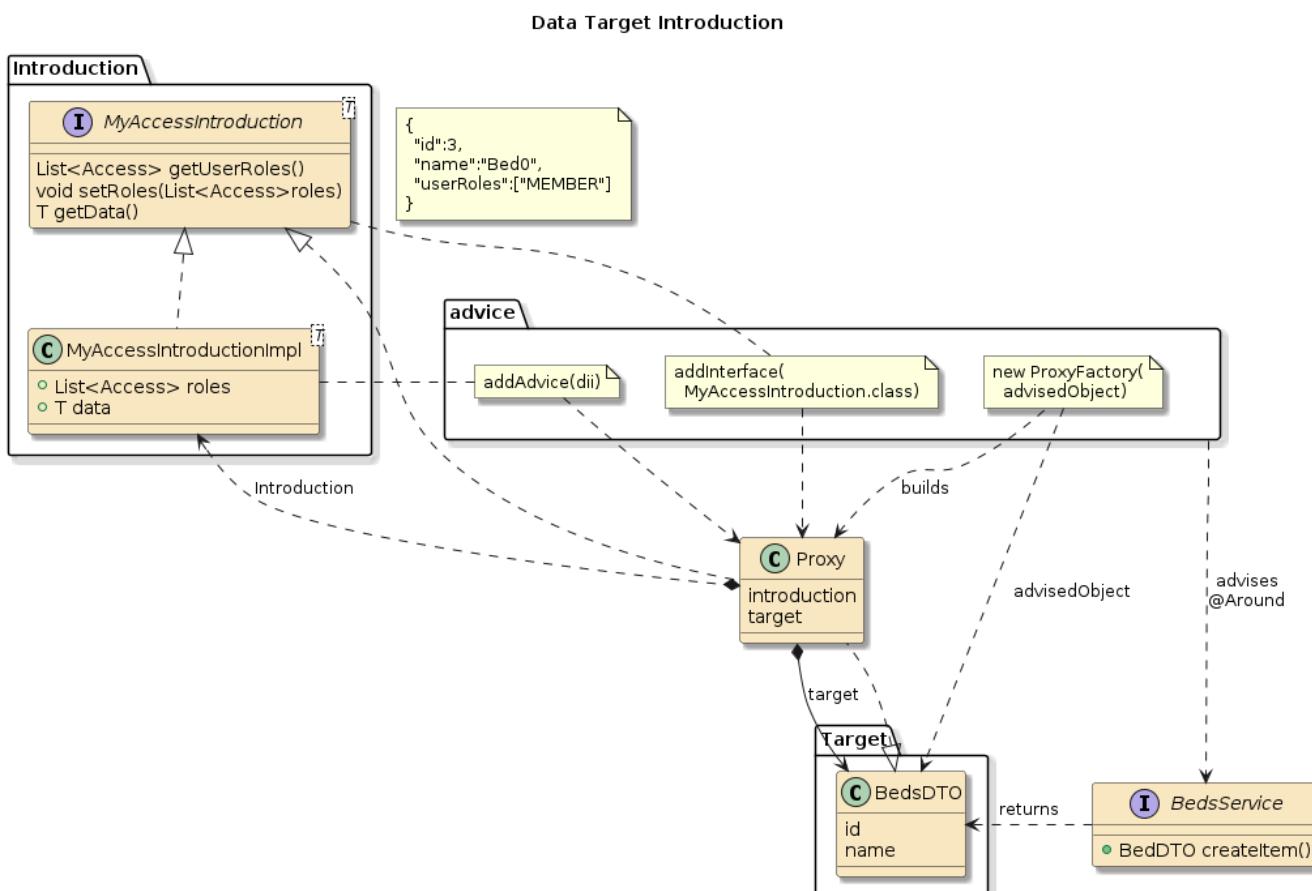


Figure 119. Data Target Introduction

The built proxy is returned to the caller in place of the target with the ability to:

- be the target **BedDTO** type with target properties
- be the **MyAccessIntroduction** type with access properties and have access to the target

The proxy should then be able to give us a mixin view of our target that will look something like the following:

Mixin View of Target Object

```
{"id":3,"name":"Bed0", ①  
 "userRoles":["MEMBER"]} ②
```

① target data object state

② Introduction state

253.2.3. Adding Introduction to Target Data Object

With pointcut defined ...

Review: Example Pointcut for Intercepting Data Target

```
@Pointcut("execution(* info.ejava.examples.svc.aop.items.services.*.getItem(..))")  
public void getter() {}
```

... we can write **@Around** advice that will use the AOP **ProxyFactory** to create a proxy for the target data object and return the proxy to the caller. This needs to be **@Around** advice since the response object will be replaced with the proxy.

Programmatically Applying Introduction to Data Target

```
@Around(value = "getter()")  
public Object decorateWithAccesses(ProceedingJoinPoint pjp) throws Throwable {  
    ItemDTO advisedObject = (ItemDTO) pjp.proceed(pjp.getArgs()); ①  
  
    //build the proxy with the target data  
    ProxyFactory proxyFactory = new ProxyFactory(advisedObject);  
    //directly support an interface for the target object  
    proxyFactory.setProxyTargetClass(true); ②  
  
    //assign the mixin  
    proxyFactory.addInterface(MyAccessIntroduction.class); ③  
    DelegatingIntroductionInterceptor dii = new DelegatingIntroductionInterceptor(new  
        MyAccessIntroductionImpl(advisedObject)); ④  
    proxyFactory.addAdvice(dii); ⑤  
  
    //return the advised object  
    ItemDTO proxyObject = (ItemDTO) proxyFactory.getProxy(); ⑥  
    return proxyObject;
```

```
}
```

- ① called method produces the target data object
- ② configure proxy to implement the interface of the target data object
- ③ configure proxy to implement the interface for the Introduction
- ④ interceptor will be a per-target Introduction
- ⑤ each proxy can have many advisors/advice
- ⑥ caller will get the proxy—wrapping the target data object—that now includes the Introduction

253.2.4. Applying Accesses to Introduction

With the Introduction in place, we can now independently assign the attributed user accesses for the specific target object. This is a fake example—so the `deriveAccess` is being used to pick access based on ID alone.

Using Introduction State in Downstream Advice

```
@AfterReturning(value = "getter()", returning = "protectedObject") ①
public void assignAccess(ItemDTO protectedObject) throws Throwable { ①
    log.info("determining access for {}", protectedObject);
    MyAccessIntroduction.Access access = deriveAccess(protectedObject.getId());

    //assigning roles
    ((MyAccessIntroduction) protectedObject).setUserRoles(List.of(access)); ②

    log.info("augmented item {} with accesses {}", protectedObject,
            (MyAccessIntroduction) protectedObject.getUserRoles());
}

//simply make up one of the available accesses for each call based on value of id
private MyAccessIntroduction.Access deriveAccess(int id) {
    int index = id % MyAccessIntroduction.Access.values().length;
    return MyAccessIntroduction.Access.values()[index];
}
```

① injecting the returned target data object as `ItemDTO` to obtain ID

② using Introduction behavior to decorate with accesses

At this point in time, the caller of `getItem()` should receive a proxy wrapping the specific target data object decorated with accesses associated with the attributed user. We are now ready for that caller to complete the end-to-end flow before we come back and discuss some additional details glossed over.

253.2.5. Using Data Introduction

The following provides a simplified version of the example client that obtains a `BedDTO` from an advised `getItem()` call and is able to have access to the Introduction state for that target data object.

Caller Obtains Target Data Object with Introduction State Decoration

```
@Autowired  
private ItemsService<BedDTO> bedsService; ①  
...  
BedDTO bed = ...  
  
BedDTO retrievedBed = bedsService.getItem(bed.getId()); ②  
  
...=(MyAccessIntroduction) retrieved).getUserRoles(); ③
```

① Spring component with advice

② advice applied to target `BedDTO` data object returned.

③ caller has access to Introduction state within target data object proxy

With the end-to-end shown, let's go back and fill in a few details.

253.2.6. Order Matters

I separated the Introduction advice into separate callbacks in order to make the logic more modular and to make a point about deterministic order. Of course, order matters when applying the related advice described above. As previously presented, we had no deterministic control over the `@AfterReturning` advice running before or after the `@Around` advice. It would not work if the Introduction was not added to the target data object until after the accesses were calculated.

Order of advice cannot be controlled using the AOP declarative technique. However, we can separate them into two (2) separate `@Aspects` and define the order at the `@Aspect` level. The snippet below shows the two (2) `@Advice` with their assigned `@Order`.

```
public class MyAccessAspects {  
    @Pointcut("execution(* info.ejava.examples.svc.aop.items.services.*.getItem(..))")  
    public void getter() {}  
  
    @Component  
    @Aspect  
    @Order(1) //run this before value assignment aspect  
    static class MyAccessDecorationAspect {  
        @Around(value = "getter()")  
        public Object decorateWithAccesses(ProceedingJoinPoint pjp) throws Throwable {  
            ...  
        }  
    }  
  
    @Component  
    @Aspect  
    @Order(0) //run this after decoration aspect  
    static class MyAccessAssignmentAspect {  
        @AfterReturning(value = "getter()", returning = "protectedObject")  
        public void assignAccess(ItemDTO protectedObject) throws Throwable {  
            ...  
        }  
    }  
}
```

With the above use of separating the advice into separate `@Aspect`, we now have deterministic control of the order of processing.

253.2.7. Jackson JSON Marshalling

One other option I wanted to achieve was to make the resulting object seamlessly appear to gain additional state for marshalling to external clients. One would normally do that making the following Jackson call using the proxy.

Proxy Marshalled to JSON

```
String json = new ObjectMapper().writeValueAsString(retrievedBed);
log.info("json={}", json);
```

The result would ideally be JSON that contained *Target Data Object State Augmented with* the target data object state augmented with *Introduction State* some Introduction state.

```
{"id":3,"name":"Bed0", ①
  "userRoles":["MEMBER"]} ②
```

① target data object state

② Introduction state

However, as presented, we will encounter this or some equally bad error. This is because Jackson and CGLIB do not play well together.

```
com.fasterxml.jackson.databind.exc.InvalidDefinitionException: Direct self-reference
leading to cycle (through reference chain:
info.ejava.examples.svc.aop.items.dto.BedDTO$$SpringCGLIB$$0["advisors"]-
>org.springframework.aop.support.DefaultIntroductionAdvisor[0]-
>org.springframework.aop.support.DefaultIntroductionAdvisor["classFilter"])
```

While researching, I read where Byte Buddy (used by Hibernate), provides better support for this type of use. An alternative I was able to get to work was to apply `@JsonSerialize` to the Introduction interface to supply a marshalling definition for Jackson to follow.

The snippet below shows where the raw target data object will be made available to Jackson using a wrapper object. That is because any attempt by any method to return the raw data target will result in `ProxyFactory` wrapping that in a proxy—which would put us back to where we started. Therefore, a wrapper record has been defined to hide the target data object from `ProxyFactory` but supply it to Jackson with the instruction to ignore it—using `@JsonUnwrapped`.

Jackson Serialization Definition for Introduction

```
@JsonSerialize(as=MyAccessIntroduction.class)
public interface MyAccessIntroduction<T> {
  ...
  @JsonIgnore ④
```

```

T getData(); ②

@JsonUnwrapped ⑤
TargetWrapper<T> getRawTarget(); ③

record TargetWrapper<T>(@JsonUnwrapped T data){} ①
}

```

- ① record type defined to wrap raw target — to hide from proxying code
- ② method returning raw target will be turned into CGLIB proxy before reaching caller when called
- ③ method returning wrapped target will provide access to raw target through record
- ④ CGLIB proxy reaching client is not compatible with Jackson - make Jackson skip it
- ⑤ Jackson will ignore the wrapper record and marshal the target contents at this level

Raw Target and Wrapped Target Access

```

public class MyAccessIntroductionImpl<T> implements MyAccessIntroduction<T> {
    ...
    @Override
    public T getData() {
        return this.data;
    }
    @Override
    public TargetWrapper<T> getRawTarget() {
        return new TargetWrapper<>(this.data);
    }
}

```

With the above constructs in place, we are able to demonstrate adding "mixin" state/behavior to target data objects. A complication to this goal was to make the result compatible with Jackson JSON. A suggestion found for the Jackson/CGLIB issue was to replace the proxy code with Byte Buddy—which happens to be what Hibernate uses for its data entity proxy implementation.

253.3. JSON Output Result

The snippet below shows the JSON payload result containing both the target object and Introduction state.

Jackson JSON Output

```

{
    ②
    "userRoles" : [ "ADMIN" ],
    ①
    "id" : 2,
    "name" : "Bed0"
}

```

① wrapped target marshalled without an element wrapper

② Introduction state marshalled with target object state

Chapter 254. Other Features

We have covered a lot of capabilities in this chapter and likely all you will need. However, know there was at least one topic left unaddressed that I thought might be of interest in certain circumstances.

- [Schema Based AOP Support](#) - Spring also offers a means to express AOP behavior using XML. They are very close in capability to what was covered here—so if you need the ability to flexibly edit aspects in production without changing the Java code—this is an attractive option.

Chapter 255. Summary

In this module, we learned:

- how we can decouple potentially cross-cutting logic from business code using different levels of dynamic invocation technology
- to obtain and invoke a method reference using Java Reflection
- to encapsulate advice within proxy classes using interfaces and JDK Dynamic Proxies
- to encapsulate advice within proxy classes using classes and CGLIB dynamically written subclasses
- to integrate Spring AOP into our project
- to programmatically construct a proxy using Spring AOP [ProxyFactory](#) that will determine proxy technology to use
- to identify method join points using AspectJ language
- to implement different types of advice (before, after (completion, exception, finally), and around)
- to inject contextual objects as parameters to advice
- to implement disparate "mixin" behavior with Introductions
 - for components using [@DeclaredParent](#)
 - for data objects using programmatic [ProxyFactory](#)

After learning this material, you will surely be able to automatically envision the implementation techniques used by Spring to add framework capabilities to our custom business objects. Those interfaces we implement and annotations we assign are likely the target of many Spring AOP aspects, adding advice in a configurable way.

Maven Integration Test

copyright © 2024 jim stafford (jim.stafford@jhu.edu)

Chapter 256. Introduction

Most of the application functionality to date within this course has been formally tested using unit tests and `@SpringBootTest` mechanisms executed within Maven Surefire plugin. This provided us a time-efficient code, test, and debug round trip—with possibly never leaving the IDE. However, the functionality of the stand-alone application itself has only been demonstrated using ad-hoc manual mechanisms (e.g., curl). In this lecture, we will look to test a fully assembled application using automated tests and the `Maven Failsafe` and other plugins.

Although generally slower to execute, Maven integration tests open up opportunities to implement automated tests that require starting and stopping external resources to simulate a more realistic runtime environment. These may be the application server or resources (e.g., database, JMS server) the application will communicate with.

256.1. Goals

You will learn:

- the need for Maven integration tests
- how to implement a Maven integration test
- how to define and activate a reusable Maven integration test setup and execution

256.2. Objectives

At the conclusion of this lecture and related exercises, you will be able to:

1. implement a Maven Failsafe integration test in order to test a Spring Boot executable JAR running as a stand-alone process
2. dynamically generate test instance-specific property values in order to be able to run in a shared CI/CD environment
3. start server process(es) configured with test instance-specific property values
4. execute tests against server process(es)
5. stop server process(es) at the conclusion of a test
6. evaluate test results once server process(es) have been stopped
7. generalize Maven constructs into a conditional Maven parent profile
8. implement and trigger a Maven integration test using a Maven parent profile

Chapter 257. Maven Integration Test

Since we are getting close to real deployments, and we have hit unit integration tests pretty hard, I wanted to demonstrate a true integration test with the Maven integration test capabilities. Although this type of test takes more time to set up and execute, it places your application in a deployed simulation that can be tested locally before actual deployment. In this and follow-on chapters, we will use various realizations of the application and will start here with a simple Spring Boot Java application—absent of any backend resources.

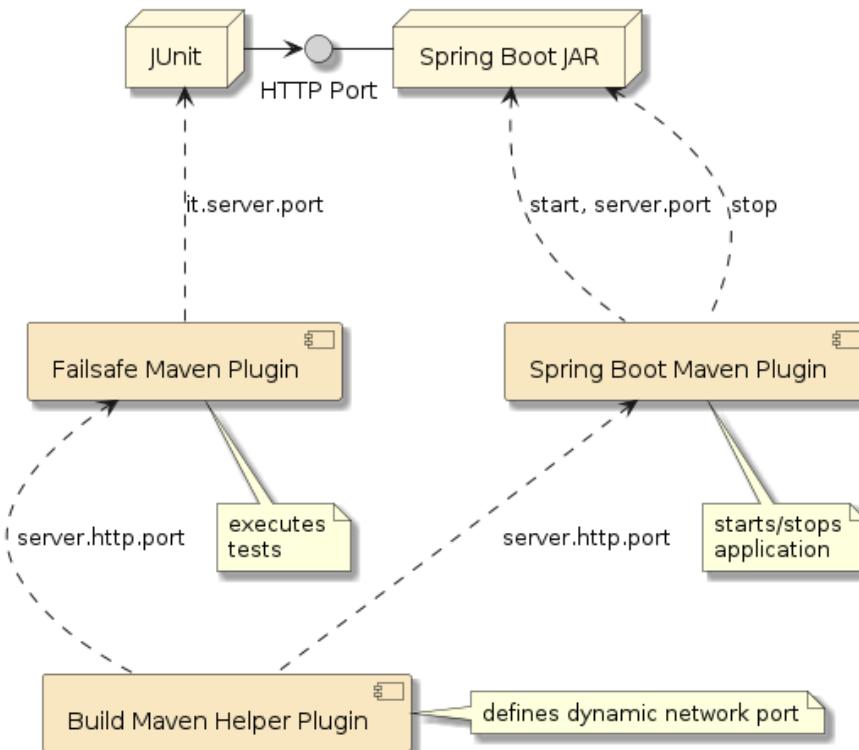


Figure 120. Maven Integration Test

A Maven integration test is very similar to the other Web API unit integration tests you are used to seeing in this course. The primary difference is that there are no server-side components in the JUnit Spring context. All the server-side components are in a separate executable (coming from the same or dependency module). The following diagram shows the participants that directly help to implement the integration test.

This will be accomplished with the aid of the Maven Failsafe, Spring Boot, and Build Maven Helper plugins.

With that said, we will still want to be able to execute integration tests like this within the IDE (i.e., start long-running server(s) and execute JUnit tests within the IDE during active test development). Therefore, expect some setup aspects to support both IDE-based and Maven-based integration testing setup in the paragraphs that follow.

257.1. Maven Integration Test Phases

Maven executes integration tests using **four (4) phases**

- **pre-integration-test** - start resources
- **integration-test** - execute tests
- **post-integration-test** - stop resources
- **verify** - evaluate/assert test results

These four (4) phases execute after the `test` phase (immediately after the `package` phase) and before the `install` phase. We will make use of three (3) plugins to perform that work within Maven:

- [spring-boot-maven-plugin](#) - used to start and stop the server-side Spring Boot process
- [build-maven-helper-plugin](#) - used to allocate a random network port for server
- [maven-failsafe-plugin](#) - used to run the JUnit JVM with the tests — passing in the `port#` — and verifying/asserting the results.

Chapter 258. Maven Integration Test Plugins

258.1. Spring Boot Maven Plugin

The `spring-boot-maven-plugin` will be configured with at least two (2) additional executions to support Maven integration testing.

1. package - (existing) builds the Spring Boot Executable JAR (bootexec)
2. **pre-integration-test** - (new) start our Spring Boot application under test
3. **post-integration-test** - (new) stop our Spring Boot application under test

The following snippet just shows the outer shell of the plugin declaration.

spring-boot-maven-plugin Outer Shell

```
<plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
  <executions>
    ...
  </executions>
</plugin>
```

① define goals, configuration, and phases for the plugin to perform

258.1.1. `spring-boot:repackage` Goal

We have already been using the `spring-boot-maven-plugin` to build an executable JAR from a Java JAR. The `build-app` execution was defined to run the `repackage` goal during the Maven `package` phase in `ejava-build-parent/pom.xml` to produce our Spring Boot Executable. The global configuration element is used to define properties that are applied to all executions. Each execution has a configuration element to define execution-specific configuration.

Pre-existing Use of Spring Boot Maven Plugin

```
<plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
  <configuration> ①
    <attach>${spring-boot.attach}</attach> ③
    <classifier>${spring-boot.classifier}</classifier> ②
  </configuration>
  <executions>
    <execution>
      <id>build-app</id>
      <phase>package</phase> ④
      <goals>
        <goal>repackage</goal> ④
      </goals>
    </execution>
  </executions>
</plugin>
```

```

    </goals>
  </execution>
</executions>
</plugin>

```

- ① global configuration applies to all executions
- ② Executable JAR will get a unique suffix identified by `classifier`
- ③ Executable JAR will not be installed or deployed in any Maven local or remote repository when `false`
- ④ `spring-boot-maven-plugin:repackage` goal executed during the Maven `package` phase with global configuration

258.1.2. `spring-boot:start` Goal

The next step is to start the built Spring Boot Executable JAR during the `pre-integration` phase — to be ready for integration tests.

The following snippet shows an example of how to configure a follow-on execution of the Spring Boot Maven Plugin to `start` the server in the background (versus a blocking `run`). The execution is configured to supply a Spring Boot `server.port` property with the HTTP port to use. We will define the port separately using the Maven `server.http.port` property at build time. The client-side `@SpringBootTest` will also need this port value for the client(s) in the integration tests.

SpringBoot pre-integration-test Execution

```

<execution>
  <id>pre-integration-test</id> ①
  <phase>pre-integration-test</phase> ②
  <goals>
    <goal>start</goal> ③
  </goals>
  <configuration>
    <skip>${skipITs}</skip> ④
    <arguments> ⑤
      <argument>--server.port=${server.http.port}</argument>
    </arguments>
  </configuration>
</execution>

```

- ① each execution must have a unique ID when there is more than one
- ② this execution will be tied to the `pre-integration-test` phase
- ③ this execution will `start` the server in the background
- ④ `-DskipITs=true` will deactivate this execution
- ⑤ `--server.port` is being assigned at runtime and used by server for HTTP listen port



skipITs support

Most plugins offer a `skip` option to bypass a configured execution and sometimes map that to a Maven property that can be expressed on the command line. [Failsafe maps their property to skipITs](#). By mapping the Maven `skipITs` property to the plugin's `skip` configuration element, we can inform related plugins to do nothing. This allows one to run the Maven `install` phase without requiring integration tests to run and pass.

The above execution phase has the same impact as if we launched the JAR manually with `--server.port`. This allows multiple IT tests to run concurrently without colliding on network port number. It also permits the use of a well-known/fixed value for use with IDE-based testing.

Manual Start Commands

```
$ java -jar target/failsafe-it-example-*-SNAPSHOT-bootexec.jar  
Tomcat started on port(s): 8080 (https) with context path '' ①  
  
$ java -jar target/failsafe-it-example-*-SNAPSHOT-bootexec.jar --server.port=7712 ②  
Tomcat started on port(s): 7712 (http) with context path '' ②
```

① Spring Boot using well-known default port#

② Spring Boot using runtime `server.port` property to override port to use

As expected, the Spring Boot Maven Plugin has first-class support for many [common Spring Boot settings](#). For example:

- arguments - as just demonstrated
- environment variables
- JVM Arguments
- profiles
- etc.

You are not restricted to just command line arguments.

258.1.3. `spring-boot:stop` Goal

Running the tests is outside the scope of the Spring Boot Maven Plugin. The next time we will need this plugin is to shut down the server during the `post-integration-phase`—after the integration tests have completed.

The following snippet shows the Spring Boot Maven Plugin being used to `stop` a running server.

SpringBoot post-integration-test Execution

```
<execution>  
  <id>post-integration-test</id> ①  
  <phase>post-integration-test</phase> ②  
  <goals>  
    <goal>stop</goal> ③
```

```

</goals>
<configuration>
    <skip>${skipITs}</skip> ④
</configuration>
</execution>

```

- ① each execution must have a unique ID
- ② this execution will be tied to the `post-integration-test` phase
- ③ this execution will `stop` the running server
- ④ `-DskipITs=true` will deactivate this execution

At this point the Spring Boot Maven Plugin is ready to do what we need as long as we define the Maven `server.http.port` property value to use. We can define that statically with a Maven pom.xml property, a `-Dserver.http.port= #####` on the Maven command line, or dynamically with the Maven Build Helper Plugin.

258.2. Build Helper Maven Plugin

The `build-helper-maven-plugin` contains `various utilities` that are helpful to create a repeatable, portable build. The snippet below shows the outer shell of the plugin declaration.

build-helper-maven-plugin Outer Shell

```

<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>build-helper-maven-plugin</artifactId>
    <executions>
        ...
    </executions>
</plugin>

```

We will populate the declaration with one execution to identify the available network port.

258.2.1. build-helper:reserve-network-port Goal

We will run the Build Helper Maven Plugin early in the lifecycle—during the `process-resources` phase so that we have it early enough to use for unit tests (`test` phase) as well as integration tests (`pre-integration-test` phase when resources are started).

We are using the `reserve-network-port` goal to select one or more random and available HTTP port numbers at build-time and assign each to a provided Maven property (using the `portName` property). The port number, in this case, is assigned to the `server.http.port` Maven property. The `server.http.port` property was shown being used as input to the Spring Boot Maven Plugin earlier.

build-helper:reserve-network-port

```

<execution>
    <id>reserve-network-port</id>

```

```

<phase>process-resources</phase> ①
<goals>
  <goal>reserve-network-port</goal> ②
</goals>
<configuration>
  <portNames> ③
    <portName>server.http.port</portName>
  </portNames>
</configuration>
</execution>

```

① execute during the `process-resources` Maven phase — which is well before `pre-integration-test`

② execute the `reserve-network-port` goal of the plugin

③ assigned the identified port to the Maven `server.http.port` property

The snippet below shows an example of the `reserve-network-port` identifying an available network port.

Example build-helper:reserve-network-port Build-time Execution

```

[INFO] --- build-helper-maven-plugin:3.4.0:reserve-network-port (reserve-network-port)
@ failsafe-it-example ---
[INFO] Reserved port 60066 for server.http.port

```

258.3. Failsafe Plugin

The [Failsafe plugin](#) is used to execute the JUnit tests. It is a near duplicate of its sibling Surefire plugin, but targeted to operate in the Maven `integration-test` phase. The following snippet shows the outer shell of the Failsafe declaration.

maven-failsafe-plugin Outer Shell

```

<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-failsafe-plugin</artifactId>
  <executions>
    ...
  </executions>
</plugin>

```

258.3.1. failsafe:integration-test Goal

The `failsafe:integration-test` goal executes the JUnit tests and automatically binds to the Maven `integration-test` phase. This execution is separate from the server started by the Spring Boot Maven Plugin but must have some common configuration in order to communicate. In the snippet below, we are adding the `integration-test` goal to Failsafe and configuring the plugin to launch the JUnit tests with an `it.server.port` property.

Failsafe integration-test Phase

```
<execution>
  <id>integration-test</id>
  <goals> ①
    <goal>integration-test</goal>
  </goals>
  <configuration>
    <systemPropertyVariables> ②
      <it.server.port>${server.http.port}</it.server.port>
    </systemPropertyVariables>
  </configuration>
</execution>
```

- ① activate integration-test goal — automatically binds to **integration-test** phase
② add a `-Dit.server.port=${server.http.port}` system property to the execution

it.server.port used in ServerConfig used by Client

`it.server.port` is used to populate the `ServerConfig @ConfigurationProperties` component within the JUnit client.



```
@Bean
@ConfigurationProperties("it.server")
public ServerConfig itServerConfig() {
    return new ServerConfig();
}
```

258.3.2. failsafe:verify Goal

The snippet below shows the final phase for Failsafe. After the integration resources have been taken down, the only thing left is to assert the results. This is performed by the **verify** Failsafe goal, which automatically binds to the Maven **verify** phase. This pass/fail assertion is delayed by a few Maven phases so that the build does not fail while integration resources are still running.

Failsafe verify Phase

```
<execution>
  <id>verify</id>
  <goals> ①
    <goal>verify</goal>
  </goals>
</execution>
```

- ① activate verify goal — automatically binds to **verify** phase



verify Goal Must Be Declared to Report Pass/Fail to Maven Build

If the **verify** goal is not wired into the declaration — the setup, tests, and tear down

will occur, but the build will not evaluate and report the results to Maven. The integration tests will appear to always pass even if you see test errors reported in the `integration-test` output.

Chapter 259. Integration Test Client

With the Maven aspects addressed, let's take a look at any subtle changes we need to make to JUnit tests running within Failsafe.

259.1. JUnit @SpringBootTest

We start with a familiar looking JUnit test and `@SpringBootTest`. We can still leverage a Spring Context, however, there is no application or server-side resources in the Spring context. The application has its Spring Context started by the Spring Boot Maven Plugin. This Spring Context is for the tests running within the Failsafe Plugin.

```
@SpringBootTest(classes=ClientTestConfiguration.class, ①
    webEnvironment = SpringBootTest.WebEnvironment.NONE) ②
public class FailsafeRestTemplateIT {
    @Autowired
    private RestTemplate authnUser;
    @Autowired
    private URI authnUrl;
```

① no application class in this integration test. Everything is server-side.

② have only a client-side web environment. No listen port necessary

259.2. ClientTestConfiguration

This trimmed down `@Configuration` class is all that is needed for JUnit test to be a client of a remote process. The `@SpringBootTest` will demand to have a `@SpringBootConfiguration` (`@SpringBootApplication` is a `@SpringBootConfiguration`) to initialize the application hosting the test. So, we will assign that annotation to our test's configuration.

We also need to add `@EnableAutoConfiguration` (normally supplied by `@SpringBootApplication`) to enable injection of external resources like `RestTemplateBuilder`.

JUnit Client SpringBootConfiguration

```
@SpringBootConfiguration(proxyBeanMethods = false) ①
@EnableAutoConfiguration ②
@Slf4j
public class ClientTestConfiguration {
    ...
    @Bean
    @ConfigurationProperties("it.server") ③
    public ServerConfig itServerConfig() { ... }
    @Bean
    public URI authnUrl(ServerConfig serverConfig) { ... } ④
    @Bean
    public RestTemplate authnUser(RestTemplateBuilder builder,...) ⑤
```

...

- ① there must be 1 `@SpringBootConfiguration` and must be supplied when running without a `@SpringBootApplication`
- ② must enable AutoConfiguration to trigger `RestTemplateBuilder` and other automatic resources
- ③ inject properties for test (e.g., `it.server.port`) from Failsafe Maven Plugin
- ④ injectable baseUrl of remote server
- ⑤ injectable `RestTemplate` with authentication and HTTP filters applied



Since Maven integration tests have no `RANDOM_PORT` and no late `@LocalServerPort` injection, bean factories for components that depend on the server port do not require `@Lazy` instantiation.

259.3. username/password Credentials

The following shows more of the `@SpringBootConfiguration` with the username and password credentials being injected using values from the properties provided a test properties file. In this test's case — they should always be provided. Therefore, no default String is defined.

username/password Credentials

```
public class ClientTestConfiguration {  
    @Value("${spring.security.user.name}")  
    private String username;  
    @Value("${spring.security.user.password}")  
    private String password;
```

259.4. ServerConfig

The following shows the primary purpose for `ServerConfig` as a `@ConfigurationProperties` class with flexible prefix. In this particular case, it is being instructed to read in all properties with prefix "it.server" and instantiate a `ServerConfig`. The Failsafe Maven Plugin is configured to supply the random port number using the `it.server.port` property.

```
@Bean  
@ConfigurationProperties("it.server")  
public ServerConfig itServerConfig() {  
    return new ServerConfig();  
}
```

The URL scheme will default to "http" and the port will default to "8080" unless a property override (`it.server.scheme`, `it.server.host`, and `it.server.port`) is supplied. The resulting value will be injected into the `@SpringBootConfiguration` class.

259.5. authnUrl URI

Since we don't have the late-injected `@LocalServerPort` for the web-server and our `ServerConfig` values are all known before starting the Spring Context, we no longer need `@Lazy` instantiation. The following shows the `baseUrl` from `ServerConfig` being used to construct a URL for "/api/authn/hello".

Building baseUrl from Injected ServerConfig

```
import org.springframework.web.util.UriComponentsBuilder;
...
@Bean
public URI authnUrl(ServerConfig serverConfig) {
    URI baseUrl = serverConfig.getBaseUrl();
    return UriComponentsBuilder.fromUri(baseUrl).path("/api/authn/hello").build()
        .toUri();
}
```

259.6. authUser RestTemplate

We are going to create separate `RestTemplate` components, fully configured to communicate and authenticate a particular way. This one and only `RestTemplate` for our example will be constructed by the `authnUser` `@Bean` factory.

By no surprise, `authnUser()` `@Bean` factory is adding a `BasicAuthenticationInterceptor` containing the injected username and password to a new `RestTemplate` for use in the test. The injected `ClientHttpRequestFactory` will take care of the HTTP/HTTPS details.

authnUser RestTemplate

```
import org.springframework.http.client.BufferingClientHttpRequestFactory;
import org.springframework.http.client.ClientHttpRequestFactory;
import org.springframework.http.client.support.BasicAuthenticationInterceptor;
import org.springframework.web.client.RestTemplate;
...
@Bean
public RestTemplate authnUser(RestTemplateBuilder builder,
                             ClientHttpRequestFactory requestFactory) {
    RestTemplate restTemplate = builder.requestFactory(
        //used to read the streams twice -- so we can use the logging filter below
        ()->new BufferingClientHttpRequestFactory(requestFactory))

        .interceptors(new BasicAuthenticationInterceptor(username, password), ①

                      new RestTemplateLoggingFilter())
        .build();
    return restTemplate;
}
```

① adding a `ClientHttpRequestInterceptor` filter, supplied by Spring to supply an HTTP BASIC

Authentication header with username and password

259.7. HTTP ClientHttpRequestFactory

The `ClientHttpRequestFactory` is supplied as a `SimpleClientHttpRequestFactory` using injection to separate the underlying transport (TLS vs. non-TLS) from the HTTP configuration.

HTTP-based ClientHttpRequestFactory

```
import org.springframework.http.client.ClientHttpRequestFactory;
import org.springframework.http.client.SimpleClientHttpRequestFactory;
...
@Bean
public ClientHttpRequestFactory httpsRequestFactory() {
    return new SimpleClientHttpRequestFactory(); ①
}
```

① an HTTP/(non-TLS)-based `ClientHttpRequestFactory`

259.8. JUnit @Test

The core parts of the JUnit test are pretty basic once we have the HTTP/Authn-enabled `RestTemplate` and `baseUrl` injected. From here it is just a normal test, except that the activity under test is remote on the server side.

```
public class FailsafeRestTemplateIT {
    @Autowired ①
    private RestTemplate authnUser;
    @Autowired ②
    private URI authnUrl;

    @Test
    public void user_can_call_authenticated() {
        //given a URL to an endpoint that accepts only authenticated calls
        URI url = UriComponentsBuilder.fromUri(authnUrl)
            .queryParam("name", "jim").build().toUri();

        //when called with an authenticated identity
        ResponseEntity<String> response = authnUser.getForEntity(url, String.class);

        //then expected results returned
        then(response.getStatusCode()).isEqualTo(HttpStatus.OK);
        then(response.getBody()).isEqualTo("hello, jim");
    }
}
```

① `RestTemplate` with authentication and HTTP aspects addressed using filters

② `authnUrl` built from `ServerConfig` and injected into test

Chapter 260. IDE Execution

JUnit tests, like all software, take a while to write and the development is more efficiently tested within the IDE. The IDE can run the JUnit test shown here just like any other test. The IDE does not care about class naming conventions. The IDE will run any class method annotated with `@Test` as a test. However, the application must be manually started (and restarted after changes to `src/main`).

There are at least two (2) ways to start the server when working with the JUnit test in the IDE. In all cases, the port should default to something well-known like `8080`.

- use the IDE to run the application class
- use the Spring Boot Maven Plugin to run the application class. The `server.port` will default to `8080`.

Using Spring Boot Maven Plugin to Run Application

```
$ mvn spring-boot:run  
...  
[INFO] --- spring-boot-maven-plugin:3.3.2:run (default-cli) @ failsafe-it-example  
---  
...  
Tomcat started on port 8080 (http) with context path '/'  
Started FailsafeExampleApp in 1.63 seconds (process running for 1.862)
```

Supply an extra `spring-boot.run.arguments` system property to pass in any custom arguments, like `port#`.

Using Spring Boot Maven Plugin to Run Application with Custom Port#

```
$ mvn spring-boot:run -Dspring-boot.run.arguments='--server.port=7777' ①  
...  
[INFO] --- spring-boot-maven-plugin:3.3.2:run (default-cli) @ failsafe-it-example  
---  
...  
Tomcat started on port(s): 7777 (http) with context path '/'  
Started FailsafeExampleApp in 2.845 seconds (process running for 3.16)
```

① use override to identify desired `server.port`; otherwise use 8080

Once the application under test by the JUnit test is running, you can then execute individual integration tests annotated with `@Test`, entire test cases, or entire packages of tests using the IDE.

Once the integration tests are working as planned, we can automate our work using Maven and the plugins we configured earlier.

Chapter 261. Maven Execution

Maven can be used to automatically run integration tests by automating all external resource needs.

261.1. Maven Verify

When we execute `mvn verify` (with an option to add `clean`), we see the port being determined and assigned to the `server.http.port` Maven property.

Starting Maven Build

```
$ mvn verify
...①
- build-helper-maven-plugin:3.4.0:reserve-network-port (reserve-network-port) @
failsafe-it-example ---
[INFO] Reserved port 52024 for server.http.port
...②
- maven-surefire-plugin:3.3.2:test (default-test) @ failsafe-it-example ---
...
- spring-boot-maven-plugin:3.3.2:repackage (build-app) @ failsafe-it-example ---
...③
- spring-boot-maven-plugin:3.3.2:start (pre-integration-test) @ failsafe-it-example
---
...
...
```

① the port identified by build-helper-maven-plugin as `52024`

② Surefire tests firing at an earlier `test` phase

③ server starting in the `pre-integration-test` phase



mvn install also runs verify

The Maven `verify` goal is right before `install`. Calling `mvn clean install` is common and runs all Surefire tests and Failsafe integration tests.

261.1.1. Server Output

When the server starts, we can see that the default profile is active and Tomcat was assigned the `52024` port value from the build.

Server Output

```
FailsafeExampleApp#logStartupProfileInfo:634 No active profile set, falling back to 1
default profile: "default" ①
TomcatWebServer#initialize:108 Tomcat initialized with port(s): 52024 (http)②
```

① no profile has been activated on the server

② server HTTP port assigned to 52024

261.1.2. JUnit Client Output

When the JUnit client starts, we can see that the baseURL contains `http` and the dynamically assigned port 52024.

JUnit Client Output

```
FailsafeRestTemplateIT#logStartupProfileInfo:640 No active profile set, falling back  
to 1 default profile: "default" ①  
ClientTestConfiguration#authnUrl:42 baseUrl=http://localhost:52024 ②  
FailsafeRestTemplateIT#setUp:30 baseUrl=http://localhost:52024/api/authn/hello
```

① no profile is active in JUnit client

② baseUrl is assigned `http` and port 52024, with the latter dynamically assigned at build-time

261.1.3. JUnit Test DEBUG

There is some DEBUG logged during the activity of the test(s).

Message Exchange

```
GET /api/authn/hello?name=jim, headers=[accept:"text/plain, application/json,  
application/xml, text/xml, application/*+json, application/*+xml, */*,  
authorization:"Basic dXNlcjpwYXNzd29yZA==", user-agent:"masked",  
host:"localhost:52024", connection:"keep-alive"]]
```

261.1.4. Failsafe Test Results

Test results are reported.

Failsafe Test Results

```
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 2.168 s -- in  
info.ejava.examples.svc.https.FailsafeRestTemplateIT  
Results:  
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

261.1.5. Server is Stopped

Server is stopped.

Server is Stopped

```
- spring-boot-maven-plugin:3.3.2:stop (post-integration-test) @ failsafe-it-example  
---  
Stopping application...  
15:44:26.510 RMI TCP Connection(4)-127.0.0.1 INFO
```

```
XBeanRegistrar$SpringApplicationAdmin#shutdown:159 Application shutdown requested.
```

261.1.6. Test Results Asserted

Test results are asserted.

Overall Test Results

```
[INFO] --- maven-failsafe-plugin:3.3.2:verify (verify) @ failsafe-it-example ---
[INFO] -----
[INFO] BUILD SUCCESS
```

Chapter 262. Maven Configuration Reuse

The plugins can often become a mouthful to completely define, and we would like to conditionally reuse that definition and activate them in sibling modules using Maven integration tests. We can define them in the parent pom and make them optionally activated within [Maven profiles](#) and activated using various [activation conditions](#).

- JDK version
- OS
- Property presence and/or value
- Packaging
- File presence
- manual activation (`-P profile_name`) or deactivation (`-P !profile_name`)

One or more conditions can be defined. Since [Maven 3.2.2](#), all conditions have been required to be satisfied to activate the profile.



Maven Profile Activation Criteria is ANDed

Maven profile activation lacks expressive predicate logic. There is no option for **OR**.



Maven Profile is Separate from Spring Profile

Both Spring and Maven have the concept of a "profile". The two are similar in concept but totally independent of one another. Activating a Maven profile does not inherently activate a Spring profile and vice versa.



Bash Requires bang("!) to be Escaped

Linux/bash shells require a bang(!) character to be escaped.

```
$ mvn clean install -P \!it
```

262.1. it Maven profile

The following snippet shows the Shell of an `it` Maven profile being defined to activate when the file `application-it.properties` appears in the `src/test/resources` directory. We can place the plugin definitions from above into the profile. We can also include other properties specific to the profile.

```
<profiles>
  <profile>
    <id>it</id> ①
    <activation>
      <file> ②
        <exists>${basedir}/src/test/resources/application-
it.properties</exists>
```

```

</file>
</activation>
<properties>
  ...
</properties>
<build>
  <pluginManagement>
    <plugins>
      ...
    </plugins>
  </pluginManagement>

  <plugins> <!-- activate the configuration -->
  ...
</plugins>
</build>
</profile>
</profiles>

```

① defining profile named **it**

② profile automatically activates when the specific file exists

262.2. Failsafe HTTPS/IT Profile Example

I have added a second variant in a separate Maven Failsafe example module ([failsafe-https-example](#)) that leverages the Maven **it** profile in [ejava-build-parent](#). The Maven profile activation only cares that the `src/test/resources/application-it.properties` file exists. The contents of the file is of no concern to the Maven **it** profile itself. The contents of `application-it.properties` is only of concern to the Spring **it** profile (when activated with `@ActiveProfiles`).

262.2.1. `it.properties` Maven Trigger File

Presence of File Activates Profile

```

<profile>
  <id>it</id>
  <activation>
    <file>
      <exists>${basedir}/src/test/resources/application-it.properties</exists>
    </file>
  </activation>

  |-- pom.xml
  '-- src
    ...
    '-- test
      ...
        '-- resources
          '-- application-it.properties ①

```

① file presence (empty or not) triggers Maven profile

262.2.2. Augmenting Parent Definition

This `failsafe-https-example` example uses a Spring `https` profile for the application and requires the Spring Boot Maven Plugin to supply that Spring profile when the application is launched. We can augment the parent Maven plugin definition by specifying the plugin and execution ID with augmentations.

Specific options for `configuration` can be found in the [Spring Boot Maven Plugin run goal documentation](#)

Augmenting Plugin Definition with Profile Configuration

```
<build>
  <plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <executions>
      <execution>
        <id>pre-integration-test</id>
        <configuration> ①
          <profiles>https</profiles>
        </configuration>
      </execution>
    </executions>
  </plugin>
```

① child pom augmenting parent configuration definition



Additional Parent Plugin Override/Customization

We have some control over `override` versus `append` using `combine.children` and `combine.self` attributes of the configuration.

262.2.3. Integration Test Properties

The later `failsafe-https-example` will require a few test properties to be defined. We can use the `application-it.properties` Maven profile activation file trigger to define them. You will recognize them from the HTTPS Security lecture.

application-it.properties

```
it.server.scheme=https
#must match self-signed values in application-https.properties
it.server.trust-store=${server.ssl.key-store}
it.server.trust-store-password=${server.ssl.key-store-password}
#used in IDE, overridden from command line during failsafe tests
it.server.port=8443
```

Same Basic HTTPS Setup Demonstrated with Surefire in the HTTPS Security Chapter



The example also requires the client Spring Context to be capable of establishing a TLS connection. This is exactly as shown in the HTTPS Security chapter and will not be shown here. See the [failsafe-https-example](#) project for the complete example these excerpts were taken from.

262.2.4. Activating Test Profiles

The JUnit test now activates two profiles: [https](#) and [it](#).

- the [https](#) profile is used to define the server communication properties that [application-it.properties](#) uses for templated property expansion. The client does not use these directly.
- the [it](#) profile is used to define the client communication properties.

Since the client property values are defined in terms of server properties, both profiles must be active to bring in both sources.

Activate Server HTTPS Properties and Client Profile

```
@SpringBootTest(classes=ClientTestConfiguration.class,
               webEnvironment = SpringBootTest.WebEnvironment.NONE)
//https profile defines server properties used here by client
//it profile defines client-specific properties
@ActiveProfiles({"https","it"}) ① ②
public class HttpsRestTemplateIT {
```

① [https](#) profile activates server properties for client IT test context to copy

② [it](#) profile defines client IT test properties templated using [https](#) defined properties

262.2.5. Example HTTPS Failsafe Test Execution

The following [mvn verify](#) output shows the complete build for the [failsafe-https-example](#) project which leverages the reusable [it](#) profile form the build parent.

Summary of Full Maven verify Build

```
$ mvn verify
- maven-resources-plugin:3.3.1:resources (default-resources)
- build-helper-maven-plugin:3.4.0:reserve-network-port (reserve-network-port)
Reserved port 59871 for server.http.port
- maven-compiler-plugin:3.11.0:compile (default-compile) @ https-hello-example ---
- maven-compiler-plugin:3.11.0:testCompile (default-testCompile) @ https-hello-example
---
- maven-surefire-plugin:3.1.2:test (default-test) @ https-hello-example ---
- maven-jar-plugin:3.3.0:jar (default-jar) @ https-hello-example ---
- maven-source-plugin:3.3.0:jar (attach-sources) @ https-hello-example ---
- spring-boot-maven-plugin:3.3.2:start (pre-integration-test) @ https-hello-example
---
...
```

```
HttpsExampleApp#logStartupProfileInfo:640 The following 1 profile is active: "https"
TomcatWebServer#initialize:108 Tomcat initialized with port(s): 59871 (https)
StandardService#log:173 Starting service [Tomcat]
...
- maven-failsafe-plugin:3.1.2:integration-test (integration-test)
...
Running info.ejava.examples.svc.https.HttpsRestTemplateIT
...
HttpsRestTemplateIT#logStartupProfileInfo:640 The following 2 profiles are active:
"https", "it"
ClientTestConfiguration#authnUrl:55 baseUrl=https://localhost:59871
...

[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
- spring-boot-maven-plugin:3.3.2:stop (post-integration-test)
[INFO] Stopping application...
- maven-failsafe-plugin:3.1.2:verify (verify)
-----
[INFO] BUILD SUCCESS
```

Chapter 263. Summary

In this module, we learned:

- the need for Maven integration tests
- implement a Maven Failsafe integration test in order to test a Spring Boot executable JAR running as a stand-alone process
 - dynamically generate test instance-specific property values in order to be able to run in a shared CI/CD environment
 - start server process(es) configured with test instance-specific property values
 - execute tests against server process(es)
 - stop server process(es) at the conclusion of a test
 - evaluate test results once server process(es) have been stopped
- how to define reusable Maven integration test setup and execution
 - generalize Maven constructs into a conditional Maven parent profile
 - implement and trigger a Maven integration test using a Maven parent profile

Unresolved directive in jhu784-notes.adoc - include::/builds/ejava-javaee/ejava-springboot-docs/courses/jhu784-notes/target/resources/docs/asciidoc/docker-{it-notes}.adoc[leveloffset=0]

Docker Integration Testing

copyright © 2024 jim stafford (jim.stafford@jhu.edu)

Chapter 264. Introduction

We have just seen how we can package our application within a Docker image and run the image in a container. It is highly likely that the applications you develop will be deployed in production within a Docker container and you will want to test the overall packaging. Before we add any external resource dependencies, I want to cover how we can automate the build and execution to perform a heavy-weight Maven Failsafe integration test against this image versus the Spring Boot executable JAR alone. The purpose of this could be to automate a test of how you are defining and building your Docker image with the application — as close as deployment as possible.

One unique aspect at the end of this lecture is coverage of developing a test to operate within a Docker-based CI/CD environment. Running a test within your native environment where localhost (relative to the IT test) is also the Docker host (where containers are running) is one thing. However, CI/CD test environments commonly run builds within Docker containers where localhost is not the Docker host. We must be aware of and account for that.

264.1. Goals

You will learn:

- to automate the build of a Docker image within a module
- to implement a heavyweight Maven Failsafe integration test of that Docker image
- to implement a Docker integration test that can run concurrently with other tests of the same module on the same Docker host
- to implement a Docker integration test that can run outside of and within a Docker-based CI/CD environment

264.2. Objectives

At the conclusion of this lecture and related exercises, you will be able to:

1. configure a Maven pom to automatically build a Docker image using a Dockerfile
2. configure a Maven pom to automatically manage the start and stop of that Docker image during a Maven integration test
3. configure a Maven pom to uniquely allocate and/or name resources so that concurrent tests can be run on the same Docker host
4. identify the hostname of the Docker host running the Docker containers
5. configure a Docker container and IT test to communicate with a variable Docker host

Chapter 265. Disable Spring Boot Plugin Start/Stop

In a previous lecture, we introduced the Maven `it` profile and how to structure it so that it would activate the Maven integration test infrastructure we needed to test a standalone Spring Boot application with a dynamically assigned server port#. In this lecture, we are using a Docker image versus a Spring Boot executable JAR. Everything else is still the same, so we would like to make use of the Maven `it` profile but turn off the start/stop of the Spring Boot executable — and start/stop our Docker container instead.

it Profile Activation Trigger

```
src/test/resources  
|-- application-it.properties ①  
...
```

① triggers activation of `it` profile

We will need to disable the start/stop of the native Spring Boot executable JAR, because we cannot have that process and our Docker container allocating the same port number for the test. To disable the `start/ stop` of the Spring Boot executable JAR and allow the Docker container to be the target of the test, we can add a few properties to cause the plugin to "skip" those goals.

Disable Spring Boot Plugin run/stop

```
<properties>  
    <!-- turn off launch of unwrapped Spring Boot server during integration-test;  
        using Docker instead -->  
    <spring-boot.run.skip>true</spring-boot.run.skip> ①  
    <spring-boot.stop.skip>true</spring-boot.stop.skip> ②
```

① property triggers Spring Boot Maven plugin to skip the start goal and leave the `${server.http.port}` unallocated

② property triggers Spring Boot Maven plugin to skip the unnecessary stop goal

If we stopped there, the `${server.http.port}` port would still be identified and our IT test would be executed by failsafe and fail because we have no server yet listening on the test port.

IT Connection Error

```
[ERROR] Errors:  
[ERROR]   DockerHelloIT.can_authenticate_with_server:59 » ResourceAccess I/O error on  
GET request for "http://localhost:59416/api/hello": Connection refused  
[ERROR]   DockerHelloIT.can_contact_server:46 » ResourceAccess I/O error on GET  
request for "http://localhost:59416/api/hello": Connection refused
```

Lets work on building and managing the execution of the Docker image.

Chapter 266. Docker Maven Plugin

A google search will come up with [several Maven plugins](#) designed to manage the building and execution of a Docker image. However, many of them are end-of-life and no longer maintained. I found one ([io.fabric8:docker-maven-plugin](#)) in 2024 that has current activity and the features we need to accomplish our goals.

The following snippet shows the outer shell of the Docker Maven Plugin. We have 2 to 3 ways to fill in the configuration details: XML, properties, and a combination of both. Both will require defining portions of the `image` element of `configuration.images`.

io.fabric8:docker-maven-plugin Outer Shell

```
<properties>
...
</properties>

<build>
  <plugins>
    <plugin>
      <groupId>io.fabric8</groupId>
      <artifactId>docker-maven-plugin</artifactId>
      <configuration> ①
        <images>
          <image>
            ...
            </image>
          </images>
        </configuration>
        <executions> ②
          ...
        </executions>
      </plugin>
    ...
  </plugins>
</build>
```

① defines how to build the image and run the image's container

② defines which plugin goals will be applied to the build

266.1. Executions

We can first enable the goals we need for the Docker Maven Plugin. The `start` and `stop` goals automatically associate themselves with the `pre/post-integration-test` phases. I have assigned the building of the Docker image to the `package` build phase, which fires just before `pre-integration-test`.

Enable Docker Maven Plugin Goals

```
<build>
  <plugins>
```

```

<plugin>
  <groupId>io.fabric8</groupId>
  <artifactId>docker-maven-plugin</artifactId>
  ...
  <executions>
    <execution>
      <id>build-image</id>
      <phase>package</phase> ①
      <goals>
        <goal>build</goal>
      </goals>
    </execution>
    <execution>
      <id>start-container</id>
      <goals> ②
        <goal>start</goal>
      </goals>
    </execution>
    <execution>
      <id>stop-container</id>
      <goals> ③
        <goal>stop</goal>
      </goals>
    </execution>
  </executions>
</plugin>

```

① `build` goal must be manually bound to the `package` phase

② `start` goal is automatically bound to the `pre-integration` phase

③ `stop` goal is automatically bound to the `post-integration` phase

If we stopped here, we would have the Docker Maven Plugin activating the `build`, `start`, and `stop` goals when we need them, but without any configuration — they will do nothing but activate when we configured them to.

Docker Maven Plugin Goals Activating

```

[INFO] --- docker-maven-plugin:0.43.4:build (build-image) @ docker-hello-example ---
...
[INFO] --- docker-maven-plugin:0.43.4:start (start-container) @ docker-hello-example
---
...
[INFO] --- docker-maven-plugin:0.43.4:stop (stop-container) @ docker-hello-example ---

```

266.2. Configuring with Properties

The Docker Maven Plugin XML configuration approach is more expressive, but the properties approach is more concise. I will be using the properties approach to take advantage of the more concise definition. We can use the plugin-default property names that start with `docker`. However, I want to highlight which properties are for my docker plugin configuration and will use a `my.docker`

prefix as show in the snippet below.

Configure Use of Property Overrides

```
<properties>
    <!-- configure the fabric8:docker plugin using properties-->
    <my.docker.verbose>true</my.docker.verbose> ②

...
<build>
    <plugins>
        <plugin>
            <groupId>io.fabric8</groupId>
            <artifactId>docker-maven-plugin</artifactId>
            <configuration>
                <images>
                    <image>
                        <external>
                            <type>properties</type> ①
                            <prefix>my.docker</prefix>
                            <mode>override</mode>
                        </external>
                    </image>
                </images>
            </configuration>
        </plugin>
    </plugins>
</build>
```

① plugin uses default property prefix `docker`. Overriding with custom `my.docker` prefix

② supplying an example `my.docker` property override

266.3. Configuration Properties

The first few properties are pretty straight forward:

(my.docker.)dockerFile

the path of the Dockerfile used to build the image. One can alternately define the all the build details within the XML elements of the `image` element if desired—but I want to stay consistent with using the Dockerfile.

(my.docker.)name

the full name:tag of the Docker image built and stored in the repository

(my.docker.)ports.api.port

external:internal port mappings that will allow us to map the internal Spring Boot 8080 port to a randomly selected external port accessible by the IT test

(my.docker.)wait.url

URL to wait for before considering the image in a running state and turning control back to the Maven lifecycle to transition from `pre-integration-test` to `integration-test`

- `server.http.port` was generated by the Build Helper Maven Plugin
- `ejava-parent.docker.hostname` will be discussed shortly. It names the Docker host, which

will be `localhost` for most development environments and different for CI/CD builds.

Example Configuration Properties

```
<properties>
  ...
  <my.docker.dockerFile>
${basedir}/Dockerfile.${docker.imageTag}</my.docker.dockerFile> ①
  <my.docker.name>${project.artifactId}:${docker.imageTag}</my.docker.name>
  <my.docker.ports.api.port>${server.http.port}:8080</my.docker.ports.api.port>
  <my.docker.wait.url>http://${ejava-
parent.docker.hostname}:${server.http.port}/api/hello?name=jim</my.docker.wait.url> ②
```

① `docker.imageTag` (2 values: `execjar` and `layered`) helps reference specific Dockerfile

② `ejava-parent.docker.hostname` Maven property definition will be shown later in this lecture. You can just think `localhost` for now.

Chapter 267. Concurrent Testing

Without additional configuration, each running instance will have a basename of the artifactId and a one-up number incremented locally, starting with 1. That means that two builds running this test and using the same Docker host could collide when using the `docker-hello-example-1` name.

Default Name Pattern

IMAGE	PORTS	NAMES
<code>docker-hello-example:execjar</code>	<code>0.0.0.0:52007->8080/tcp</code>	<code>docker-hello-example-1</code>

To override this behavior, we can assign a `containerNamingPattern` to include a random number. Once we are setting the `containerNamingPattern`, we need to explicitly set the image `alias`. I am assigning it to the same `artifactId` value we saw earlier.

Plugin with Container Name Configuration

```
<build>
  <plugins>
    <plugin>
      <groupId>io.fabric8</groupId>
      <artifactId>docker-maven-plugin</artifactId>
      <configuration>
        <containerNamePattern>%a-%t</containerNamePattern> ②
        <images>
          <alias>${project.artifactId}</alias> ①
          <image>
            <external>
            ...
        
```

① define an alias that will be used to reference the Docker image

② define a container name pattern that will be used to track running container(s) of the image

With the above configuration, we can have a randomly unique name generated that still makes some sense to us when we see it in the Docker host status.

Configured Name Pattern

IMAGE	PORTS	NAMES
<code>docker-hello-example:execjar</code>	<code>0.0.0.0:52913->8080/tcp</code>	<code>docker-hello-example-1705249438850</code> ①

① random number has been assigned to artifactId to make container name unique (required) within the Docker host

Chapter 268. Non-local Docker Host

For cases where we are running CI/CD builds within a Docker container, we will not have a local Docker host accessible via localhost. That means that if our IT test uses "http://localhost:...", it will not locate the server. To do so would require running Docker within Docker (termed "[DinD](#)")—which is not a popular setting due to elevated permissions required for the CI/CD image. Instead, special settings will be applied to the CI/CD container to identify the non-local location of the Docker host (termed "[wormhole pattern](#)"). The server is running on the Docker host as a sibling of the CI/CD build image. The IT test and `my.docker.wait.url` need to reference that non-localhost value.

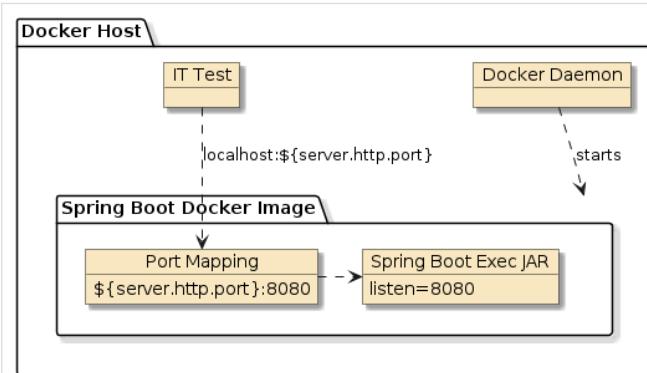


Figure 121. IT Test Running Locally in Native Environment

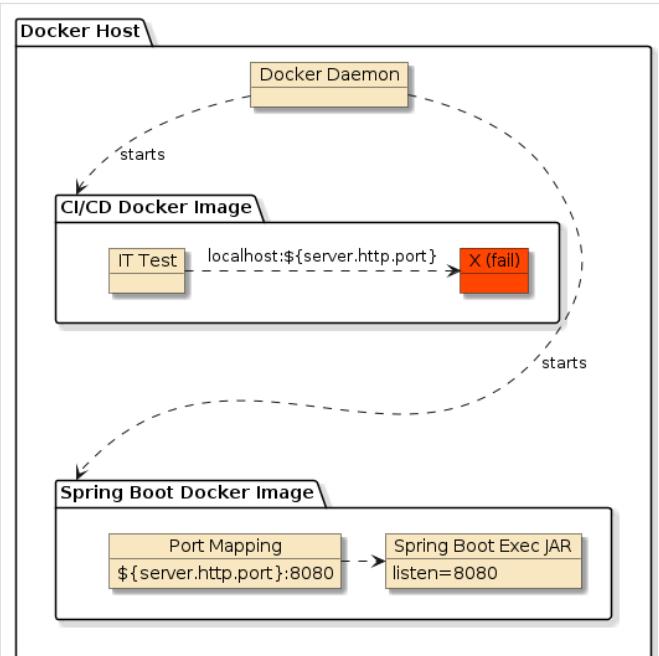


Figure 122. IT Test Running within CI/CD Docker Image

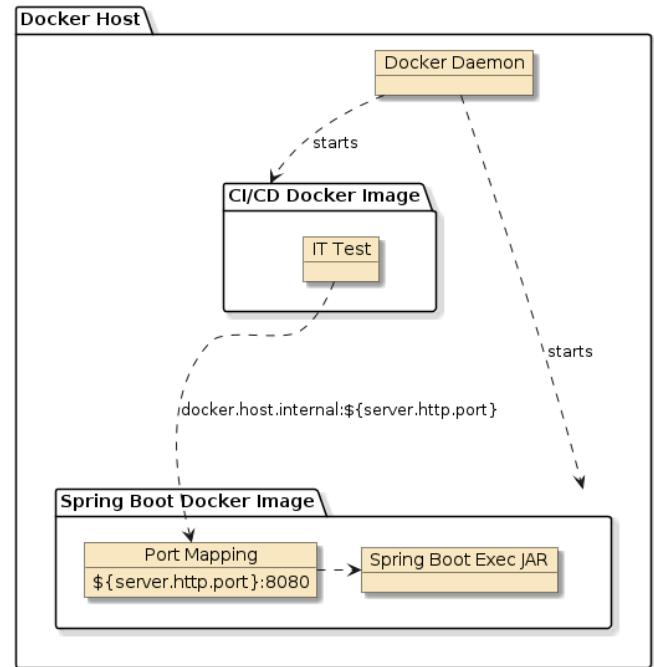
- IT Test running native on Docker host (`localhost` == Docker host)
- Spring Boot Docker image `server.http.port` is exposed on the Docker host
- IT Test locates Docker host using `localhost`

- IT Test and Spring Boot server running within sibling Docker images (`localhost` != Docker host)
- Spring Boot Docker image `server.http.port` is exposed on the Docker host
- IT test `localhost` is now within its local CI/CD Docker image and will fail to find the server running in the sibling Docker image

Tim van Baarsen wrote a [nice explanation of our problem/solution in an article](#). He states that both Windows and MacOS-based Docker installations inherently have a `host.docker.internal` hostname added to images (that does not show up in `/etc/hosts`).

Any container launched by the Docker host will have its exposed port(s) available on the Docker host and referenced by `docker.host.internal`.

- IT Test and Spring Boot server running within sibling Docker images (`localhost` != Docker host)
- Spring Boot Docker image `server.http.port` is exposed on the Docker host
- IT test uses `docker.host.internal` to locate *Figure 123. CI/CD Docker Image Configured with Docker host (`docker.host.internal` == Docker Docker Host Address host)*



The following snippet shows that even though `docker.host.internal` is not exposed in the `/etc/hosts` file, a `ping` command is able to resolve `host.docker.internal` to an IP address. This command was run on MacOS.

Resolving host.docker.internal

```
$ docker run --rm mbentley/healthbomb grep -c host.docker.internal /etc/hosts
0 ①

$ docker run --rm mbentley/healthbomb ping host.docker.internal
PING host.docker.internal (192.168.65.254): 56 data bytes ②
```

① running image on MacOS, the `host.docker.internal` name does not show in `/etc/hosts`

② using an image with `ping` command, we can show that `host.docker.internal` is resolvable

The following snippet shows a curl command resolving the `host.docker.internal` name to the Docker host where our test image is running. The curl command successfully reaches our server—which again—is not running on `localhost` relative to the curl command/client. Curl, in the case is simulating the conditions of the IT test.

Client Completes Call Using Special Hostname

```
$ docker run --rm -p 8080:8080 docker-hello-example:execjar ①
# or
$ mvn docker:run -Dserver.http.port=8080 ②
...
IMAGE          PORTS          NAMES
docker-hello-example:execjar  0.0.0.0:8080->8080/tcp  thirsty_bose
```

```
...
$ docker run --rm curlimages/curl curl
http://host.docker.internal:8080/api/hello?name=jim ③
hello, jim
```

- ① start Docker container using raw Docker command (listening on port 8080 on Docker host)
- ② start Docker container using Maven plugin (listening on port 8080 on Docker host)
- ③ Curl client running within sibling Docker image (`localhost` != Docker host; `host.docker.internal` == Docker host)

268.1. Linux Work-around

As Tim van Baarsen points out, the automatic feature provided by Docker Desktop on Windows and MacOs is not automatically provided within Linux installations. We can manually configure the execution to define a hostname with the value of the network between the Docker host and container(s)—obtained by resolving `host-gateway`. `host-gateway` is set to the IP address of the gateway put in place between the container and the Docker host.

Manually Adding host.docker.internal Using host-gateway Thru Docker Command

```
$ docker run --rm --add-host=host.docker.internal:host-gateway curlimages/curl grep
host.docker.internal /etc/hosts ①
192.168.65.254 host.docker.internal
```

- ① command line `--add-host=host.docker.internal:host-gateway` maps the `host.docker.internal` hostname to the network between the container and Docker host

The `docker-compose.yml` file provided in the root of the example source tree supplies that value using the `extra_hosts` element.

Manually Adding host.docker.internal Using host-gateway Thru Docker Compose File

```
extra_hosts:
- host.docker.internal:host-gateway
```

With the `add-host/extra_hosts` configured, we are able to resolve the Docker host in Windows, MacOS, and Linux environments.

268.2. Failsafe

Our IT test will need to know the Spring Boot server's hostname in order to properly resolve. We can configure the server's hostname using the `it.server.host` Spring Boot property in the IT test using Failsafe configuration.



it.server Properties Mapped to ServerConfig

Remember that `it.server` properties are mapped to the `ServerConfig`

`@ConfigurationProperties` instance for IT tests. This is a product of the `ejava` libraries, enabled by Spring Boot but not part of Spring Boot

The Spring Boot property can be added to the Failsafe configuration by appending to the `systemPropertyVariables`. The snippet below shows the child pom appending new properties to the parent definition (which supplied `it.server.port`). The `options` are to:

- `combine.children="append"` - add child values to parent-provided values
- `combine.self="override"` - replace parent-provided values with child values

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-failsafe-plugin</artifactId>
    <executions>
        <execution>
            <id>integration-test</id>
            <configuration> <!-- account for CD/CD environment when server will not be
localhost -->
                <systemPropertyVariables combine.children="append">
                    <it.server.host>${ejava-parent.docker.hostname}</it.server.host>
                </systemPropertyVariables>
            </configuration>
        </execution>
    </executions>
</plugin>
```

268.3. Resolving docker.hostname

We have all configurations referencing `${ejava-parent.docker.hostname}`. We now need to make sure this value is either set to `host.docker.internal` (within Docker) or `localhost` (within native environment) depending on our runtime environment.

To make this decision, I am leveraging the fact that we control the CI/CD Docker image and have knowledge of an environment variable called `TESTCONTAINERS_HOST_OVERRIDE` that exists to guide another Docker-based test tool. For this example, it does not matter what we call it as long as we know what to look for.

Root-level docker-compose.yml

```
environment:
  - TESTCONTAINERS_HOST_OVERRIDE=host.docker.internal ①
extra_hosts:
  - host.docker.internal:host-gateway ②
```

① explicitly setting a well-known environment variable within CI/CD environment

② explicitly defining `host.docker.internal` for all CI/CD environments

We can use the presence or absence of the `TESTCONTAINERS_HOST_OVERRIDE` environment variable to

provide a value for an `ejava-parent.docker.hostname` Maven build property.

Inside CI/CD Docker image

`host.docker.internal`

Outside CI/CD Docker image / Native Environment

`localhost`

```
<profile> <!-- build is running within Docker-based CI/CD via root level docker-compose.yml -->
  <id>wormhole-build</id>
  <activation>
    <property>
      <name>env.TESTCONTAINERS_HOST_OVERRIDE</name> ①
    </property>
  </activation>
  <properties> <!-- this hostname is mapped to "host-gateway", used by testcontainers, but generically usable -->
    <ejava-parent.docker.hostname>${env.TESTCONTAINERS_HOST_OVERRIDE}</ejava-parent.docker.hostname> ②
  </properties>
</profile>
<profile> <!-- build is running outside of Docker/root-level docker-compose.yml -->
  <id>native-build</id>
  <activation>
    <property>
      <name>!env.TESTCONTAINERS_HOST_OVERRIDE</name> ③
    </property>
  </activation>
  <properties> <!-- localhost outside of Docker CI/CD -->
    <ejava-parent.docker.hostname>localhost</ejava-parent.docker.hostname>④
  </properties>
</profile>
```

① we know our CI/CD container will have `TESTCONTAINERS_HOST_OVERRIDE` defined in all cases

② in our CI/CD container, environment variable `TESTCONTAINERS_HOST_OVERRIDE` will resolve to `host.docker.internal`

③ we assume the lack of `TESTCONTAINERS_HOST_OVERRIDE` means we are in native environment

④ in native environment, Docker containers should be accessible via `localhost` in normal cases



We have the option to use the `docker.host.address` property supplied by Docker Maven Plugin for cases when Docker host is truly remote and `localhost` is not correct. However, I wanted to keep this part simple and independent of the Docker Maven Plugin.

268.4. Example Output

With everything setup, we can now run our IT test against the Docker image. The Docker Compose file used in this example is at the [root of the class examples](#) tree. It hosts the ability to run Maven commands within a Docker container. We will discuss Docker Compose in a follow-on lecture.

- running locally in the native environment

Maven/IT Test Running in Native Environment

```
$ env | grep -c TESTCONTAINERS_HOST_OVERRIDE
0 ①

$ mvn clean verify -DitOnly
...
DockerHelloIT#init:34 baseUrl=http://localhost:54132 ②
RestTemplate#debug:127 HTTP GET http://localhost:54132/api/hello?name=jim
...
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
```

① `TESTCONTAINERS_HOST_OVERRIDE` system property is not present

② host defaults to `localhost`

- running within the CI/CD Docker image

Maven/IT Test Running within Docker CI/CD Environment

```
$ docker-compose -f ../../docker-compose.yml run --rm mvn env | grep
TESTCONTAINERS_HOST_OVERRIDE
TESTCONTAINERS_HOST_OVERRIDE=host.docker.internal ①

$ docker-compose -f ../../docker-compose.yml run --rm mvn mvn clean verify
-DitOnly
...
DockerHelloIT#init:34 baseUrl=http://host.docker.internal:35423 ②
RestTemplate#debug:127 HTTP GET
http://host.docker.internal:35423/api/hello?name=jim
...
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
```

① `TESTCONTAINERS_HOST_OVERRIDE` system property is present

② host is assigned to value of `TESTCONTAINERS_HOST_OVERRIDE` — `host.docker.internal`

Chapter 269. Summary

In this module, we learned:

- to automate the build of a Docker image within a module using a Maven plugin
- to implement a heavyweight integration test of that Docker image using the integration goals of a Docker plugin
- to address some singleton matters when running the Docker images simultaneously on the same Docker host.
- to configure a Docker image to communicate with another Docker image running on the same non-local Docker host. This is something common in CI/CD environments.

Docker Compose

copyright © 2024 jim stafford (jim.stafford@jhu.edu)

Chapter 270. Introduction

In a few previous lectures we have used the raw Docker API command line calls to perform the desired goals. At some early point there will become unwieldy and we will be searching for a way to wrap these commands. Years ago, I resorted to [Ant](#) and the `exec` command to wrap and chain my high level goals. In this lecture we will learn about something far more native and capable to managing Docker containers—docker-compose.

270.1. Goals

You will learn:

- how to implement a network of services for development and testing using Docker Compose
- how to operate a Docker Compose network lifecycle and how to interact with the running instances

270.2. Objectives

At the conclusion of this lecture and related exercises, you will be able to:

1. identify the purpose of Docker Compose for implementing a network of virtualized services
2. create a Docker Compose file that defines a network of services and their dependencies
3. custom configure a Docker Compose network for different uses
4. perform Docker Compose lifecycle commands to build, start, and stop a network of services
5. execute ad-hoc commands inside running images
6. instantiate back-end services for use with the follow-on database lectures

Chapter 271. Development and Integration Testing with Real Resources

To date, we have primarily worked with a single Web application. In the follow-on lectures we will soon need to add back-end database resources.

We can test with mocks and in-memory versions of some resources. However, there will come a day when we are going to need a running copy of the real thing or possibly a specific version.

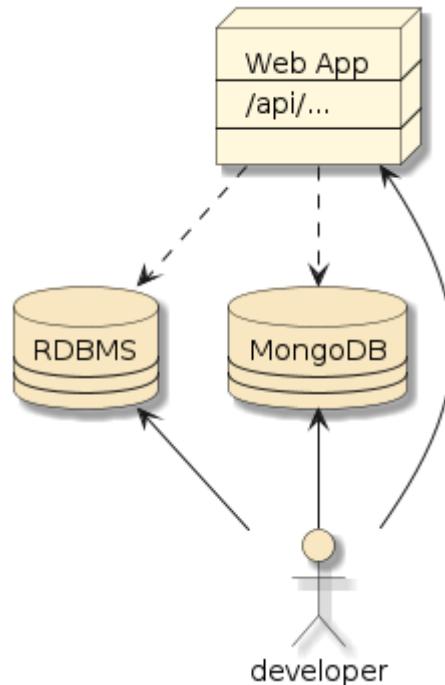


Figure 124. Need to Integrate with Specific Real Services

We have already gone through the work to package our API service in a Docker image and the Docker community has built a [plethora of offerings](#) for ready and easy download. Among them are Docker images for the resources we plan to eventually use:

- [MongoDB](#)
- [Postgres](#)

It would seem that we have a path forward.

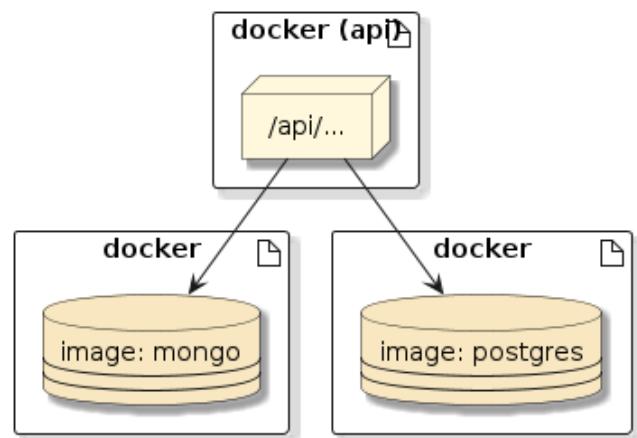


Figure 125. Virtualize Services with Docker

271.1. Managing Images

You know from our initial Docker lectures that we can easily download the images and run them individually (given some instructions) with the `docker run` command. Knowing that—we could try doing the following and almost get it to work.

Manually Starting Images

```
$ docker run --rm -p 27017:27017 \
-e MONGO_INITDB_ROOT_USERNAME=admin \
-e MONGO_INITDB_ROOT_PASSWORD=secret mongo:4.4.0-bionic ①

$ docker run --rm -p 5432:5432 \
-e POSTGRES_PASSWORD=secret postgres:12.3-alpine ②

$ docker run --rm -p 9080:8080 \
-e DATABASE_URL=postgres://postgres:secret@host.docker.internal:5432/postgres \
-e MONGODB_URI=mongodb://admin:secret@host.docker.internal:27017/test?authSource=admin \
dockercompose-hello-example:latest ③ ④
```

① using the mongo container from Dockerhub

② using the postgres container from Dockerhub

③ using an example Spring Boot Web application that simply forms database connections

④ using `host.docker.internal` since databases are running outside of the application server's "localhost" machine, on a sibling container, accessible through the Docker host

However, this begins to get complicated when:

- we start integrating the API image with the individual resources through networking
- we want to make the test easily repeatable
- we want multiple instances of the test running concurrently on the same Docker host without interference with one another

Lets not mess with manual Docker commands for too long! There are better ways to do this with Docker Compose.

Chapter 272. Docker Compose

Docker Compose is a tool for defining and running multi-container Docker applications. With Docker Compose, we can:

- define our network of applications in a single (or set of) YAML file(s)
- start/stop containers according to defined dependencies
- run commands inside of running containers
- treat the running application(s) (running within Docker container(s)) as normal

272.1. Docker Compose is Local to One Machine

Docker Compose runs everything local. It is a modest but necessary step above Docker but far simpler than any of the distributed environments that logically come after it (e.g., Docker Swarm, Kubernetes). If you are familiar with [Kubernetes](#) and [Minikube](#), then you can think of Docker Compose is a very simple/poor man's [Helm Chart](#). "Poor" in that it only runs on a single machine. "Simple" because you only need to define details of each service and not have to worry about distributed aspects or load balancing that might come in a more distributed solution.

With Docker Compose, there:

- are one or more YAML configuration files
- is the opportunity to apply environment variables and extensions (e.g., configuration specializations)
- are commands to build images and control lifecycle actions of the network

Let's start with the Docker Compose configuration file.

Chapter 273. Docker Compose Configuration File

The [Docker Compose \(configuration\) file](#) is based on [YAML](#)—which uses a concise way to express information based on indentation and firm symbol rules. Assuming we have a simple network of three (3) services, we can limit our definition to individual [services](#). The [version](#) element has been eliminated.

docker-compose.yml Shell

```
#version: '3.8' - version element eliminated
services:
  mongo:
    ...
  postgres:
    ...
  api:
    ...
```

- **version** - informs the docker-compose binary what features could be present within the file. This element has been eliminated and will produce a warning if present.
- **services** - lists the individual nodes and their details. Each node is represented by a Docker image and we will look at a few examples next.

Refer to the [Compose File Reference](#) for more details.

273.1. mongo Service Definition

The [mongo](#) service defines our instance of [MongoDB](#).

mongo Service Definition

```
mongo:
  image: mongo:4.4.0-bionic
  environment:
    MONGO_INITDB_ROOT_USERNAME: admin
    MONGO_INITDB_ROOT_PASSWORD: secret
  # ports: ①
  #   - "27017" ②
  #   - "27017:27017" ③
  #   - "37001:27017" ④
  #   - "127.0.0.1:37001:27017" ⑤
```

① not assigning port# here

② 27017 internal, random external

③ 27017 both internal and external

- ④ 37001 external and 27017 internal
- ⑤ 37001 exposed only on 127.0.0.1 external and 27017 internal

- **image** - identifies the name and tag of the Docker image. This will be automatically downloaded if not already available locally
- **environment** - defines specific environment variables to be made available when running the image.
 - VAR: X passes in variable VAR with value X.
 - VAR by itself passes in variable VAR with whatever the value of VAR has been assigned to be in the environment executing Docker Compose (i.e., environment variable or from environment file).
- **ports** - maps a container port to a host port with the syntax "host interface:host port#:container port#"
 - host port#:container port# by itself will map to all host interfaces
 - "container port#" by itself will be mapped to a random host port#
 - no ports defined means the container port# that do exist are only accessible within the network of services defined within the file.

273.2. postgres Service Definition

The `postgres` service defines our instance of Postgres.

postgres Service Definition

```
postgres:
  image: postgres:12.3-alpine
#  ports: ①
#    - "5432:5432"
  environment:
    POSTGRES_PASSWORD: secret
```

- the default username and database name is `postgres`
- assigning a custom password of `secret`

Mapping Port to Specific Host Port Restricts Concurrency to one Instance



Mapping a container port# to a fixed host port# makes the service easily accessible from the host via a well-known port# but restricts the number of instances that can be run concurrently to one. This is typically what you might do with development resources. We will cover how to do both easily — shortly.

273.3. api Service Definition

The `api` service defines our API container. This service will become a client of the two database services.

api Service Definition

```
api:  
  build: #make root ${project.basedir}  
  context: ../../..  
  dockerfile: src/main/docker/Dockerfile  
  image: dockercompose-hello-example:latest  
  ports:  
    - "${HOST_API_PORT:-8080}:8080"  
  depends_on:  
    - mongo  
    - postgres  
  environment:  
    - MONGODB_URI=mongodb://admin:secret@mongo:27017/votes_db?authSource=admin  
    - DATABASE_URL=postgres://postgres:secret@postgres:5432/postgres
```

- **build** - identifies information required to build the image for this service. If the `docker-compose.yml` and `Dockerfile` are both at the root of the module using their default (`Dockerfile`) name, these properties are not necessary.
 - **context** - Identifies a directory that all relative paths will be based. Expressing a path that points back to `${project.basedir}` makes all module artifacts available. The default is “.”.
 - **dockerfile** - defines the path and name of the Dockerfile. The default is `${context}/Dockerfile`.
- **image** - identifies the name and tag used for the image. If building, this "name:tag" will be used to name the image. Otherwise, the "name:tag" will be used to pull the image and start the container.
- **ports** - using a `${variable:-default}` reference so that we have option to expose the container port# 8080 to a dynamically assigned host port# during testing. If `HOST_API_PORT` is not resolved to a value, the default `8080` value will be used.
- **depends_on** - establishes a dependency between the images. This triggers a start of dependencies when starting this service. It also adds a hostname to this image's environment (i.e., `/etc/hosts`). Therefore, the `api` server can reach the other services using hostnames `mongo` and `postgres`. You will see an example of that when you look closely at the URLs in the later examples.
- **environment** - environment variables passed to Docker image.
 - used for supplying environment variables: e.g., `SPRING_PROFILES_ACTIVE`, `SPRING_DATASOURCE_URL`
 - if only the environment variable name is supplied, its value will not be defined here and the value from external sources will be passed at runtime
 - `DATABASE_URL` was a specific environment variable supplied by the Heroku hosting

environment — where some of this example is based upon.

273.4. Project Directory

By default, all paths in the docker-compose.yml and Docker files are relative to where the file is located. If we place our Docker-based files in a source directory, we must add a `--project-directory` reference to the Docker compose command.

Dockerfile Sources Nested in Source Tree

```
$ tree src/main/docker/ target/
src/main/docker/
|-- Dockerfile
|-- docker-compose.yml
`-- run_env.sh
target/
`-- dockercompose-it-example-6.1.0-SNAPSHOT-SNAPSHOT-bootexec.jar
```

The following shows an example execution of a `docker-compose.yml` and Dockerfile, located within the `src/main/docker` directory, making references to `./src` and `./target` directories to pick up the Spring Boot executable JAR and other resources out of the source tree.

Relative Paths are Off Module Root

```
$ egrep 'src|target' src/main/docker/*
src/main/docker/Dockerfile:COPY src/main/docker/run_env.sh . ①
src/main/docker/Dockerfile:ARG JAR_FILE=target/*-bootexec.jar ②
src/main/docker/docker-compose.yml:      dockerfile: src/main/docker/Dockerfile ①
```

① reference to `./src` tree

② reference to `./target` tree

We can make the relative paths resolve with either the `build.context` property of the `docker-compose.yml` file referencing the module root.

docker-compose build.context Setting Root for Relative Paths

```
api:
  build: #make root ${project.basedir}
  context: ../../..
```

An alternative would be to manually set the context using `--project-directory`.

Command Line Setting Root for Relative Paths

```
$ docker-compose --project-directory . -f src/main/docker/docker-compose.yml #command
```

273.5. Build/Download Images

We can trigger the build or download of necessary images using the `docker-compose build` command or simply by starting `api` service the first time.

Building API Service

```
docker-compose -f src/main/docker/docker-compose.yml build
=> [api internal] load build definition from Dockerfile
...
=> => naming to docker.io/library/dockercompose-hello-example:latest
```

After the first start, a re-build is only performed using the `build` command or when the `--build` option.

273.6. Default Port Assignments

If we start the services with `HOST_API_PORT` defined, ...

```
$ export HOST_API_PORT=1234 && docker-compose -f src/main/docker/docker-compose.yml up
-d ①
[+] Running 3/3
✓ Container docker-mongodb-1 Started
✓ Container docker-postgres-1 Started
✓ Container docker-api-1 Started
```

① `up` starts service and `-d` runs the container in the background as a daemon

Our primary interface will be the `api` service. The `api` service was assigned a variable (value `1234`) port#—which is accessible to the host’s network. If we don’t need direct `mongo` or `postgres` access (e.g., in production), we could have eliminated any host mapping. However, for development, it will be good to be able to inspect the databases directly using a mapped port.

```
docker-compose -f src/main/docker/docker-compose.yml ps
NAME                  SERVICE      STATUS          PORTS
docker-api-1          api          Up  About a minute  0.0.0.0:1234->8080/tcp
docker-mongodb-1      mongodb     Up  About a minute  0.0.0.0:27017->27017/tcp
docker-postgres-1      postgres    Up  About a minute  0.0.0.0:5432->5432/tcp
```

Our application has two endpoints `/api/hello/jdbc` and `/api/hello/mongo`, which echo a string status of the successful connection to the databases to show that everything was wired up correctly.

Using Variable-Assigned API Port#

```
$ curl http://localhost:1234/api/hello/jdbc
jdbc:postgresql://postgres:5432/postgres
```

```
$ curl http://localhost:1234/api/hello/mongo
{type=STANDALONE, servers=[{address=mongodb:27017, type=STANDALONE, roundTripTime=1.9
ms, state=CONNECTED}]}
```

273.7. Compose Override Files

Docker Compose files can be layered from base (shown above) to specialized. The following example shows the previous definitions being extended to include mapped host port# mappings. We might add this override in the development environment to make it easy to access the service ports on the host's local network using well-known port numbers.

Example Compose Override File

```
services:
  mongo:
    ports:
      - "27017:27017"
  postgres:
    ports:
      - "5432:5432"
```

Override Limitations May Cause Compose File Refactoring



There is a limit to what you can override versus augment. Single values can replace single values. However, lists of values can only contribute to a larger list. That means we cannot create a base file with ports mapped and then a build system override with the port mappings taken away.

273.8. Compose Override File Naming

Docker Compose looks for a specially named file of `docker-compose.override.yml` in the local directory next to the local `docker-compose.yml` file.

Example File Override Syntax

```
$ ls docker-compose.*  
docker-compose.override.yml docker-compose.yml  
  
$ docker-compose up ①
```

① Docker Compose automatically applies overrides from `docker-compose.override.yml` in this case

273.9. Multiple Compose Files

Docker Compose will accept a series of explicit `-f file` specifications that are processed from left to right. This allows you to name your own override files.

Example File Override Syntax

```
$ docker-compose -f docker-compose.yml -f development.yml up ①
$ docker-compose -f docker-compose.yml -f integration.yml up
$ docker-compose -f docker-compose.yml -f production.yml up
```

① starting network in foreground with two configuration files, with the left-most file being specialized by the right-most file

273.10. Environment Files

Docker Compose will look for variables to be defined in the following locations in the following order:

1. as an environment variable
2. in an environment file
3. when the variable is named and set to a value in the Compose file

Docker Compose will use `.env` as its default environment file. A file like this would normally not be checked into CM since it might have real credentials, etc.

.env Files Normally are not Part of SCM Check-in

```
$ cat .gitignore
...
.env
```

Example .env File

```
HOST_API_PORT=9090
```

The following shows an example of the `.env` file being automatically applied.

Using .env File

```
$ unset HOST_API_PORT
$ docker-compose -f src/main/docker/docker-compose.yml up -d
882a1fc54f14  dockercompose-hello-example:latest  0.0.0.0:9090->8080/tcp
```

You can also explicitly name an environment file to use. The following is explicitly applying the `alt-env` environment file — thus bypassing the `.env` file.

Example Explicit Environment File

```
$ cat alt-env
HOST_API_PORT=9999
```

```
$ docker-compose -f src/main/docker/docker-compose.yml --env-file alt-env up -d ①
$ docker ps
CONTAINER      IMAGE                               PORTS                         NAMES
5ff205dba949  dockercompose-hello-example:latest  0.0.0.0:9999->8080/tcp  docker-api-1
...
```

① starting network in background with an alternate environment file mapping API port to 9999

Chapter 274. Docker Compose Commands

274.1. Build Source Images

With the `docker-compose.yml` file defined—we can use that to control the build of our source images. Notice in the example below that it is building the same image we built in the previous lecture.

Example Docker Compose build Output

```
$ docker-compose -f src/main/docker/docker-compose.yml build
...
=> => naming to docker.io/library/dockercompose-hello-example:latest
=> [api] resolving provenance for metadata file
```

274.2. Start Services in Foreground

We can start all the services in the foreground using the `up` command. The command will block and continually tail the output of each container.

Example docker-compose up Command

```
$ docker-compose -f src/main/docker/docker-compose.yml up
[+] Running 4/3
✓ Network docker_default      Created
✓ Container docker-postgres-1 Created
✓ Container docker-mongodb-1  Created
✓ Container docker-api-1      Created
Attaching to api-1, mongodb-1, postgres-1
...
```

We can trigger a new build with the `--build` option. If there is no image present, a build will be triggered automatically but will not be automatically reissued on subsequent commands without supplying the `--build` option.

274.3. Project Name

Docker Compose names all of our running services using a project name prefix. The default project name is the parent directory name. Notice below how the parent directory name `docker-hello-example` was used in each of the running service names.

Project Name Defaults to Parent Directory Name

```
$ pwd
.../dockercompose-it-example

docker-compose --project-directory . -f src/main/docker/docker-compose.yml up -d
```

```
[+] Running 3/3
✓ Container dockercompose-it-example-postgres-1 Started
✓ Container dockercompose-it-example-mongodb-1 Started
✓ Container dockercompose-it-example-api-1 Started
```

We can explicitly set the project name using the `-p` option. This can be helpful if the parent directory happens to be something generic—like `target` or `src/test/resources`.

```
docker-compose -f src/main/docker/docker-compose.yml -p foo up -d ①
[+] Running 4/4
✓ Network foo_default      Created
✓ Container foo-postgres-1 Started ②
✓ Container foo-mongodb-1  Started
✓ Container foo-api-1     Started
```

① manually setting project name to `foo`

② network and services all have prefix of `foo`



Notice that when we set `--project-directory` as `".`, the project name is set to `dockercompose-it-example` because that is the directory name for the relative path `"."`. If we remove the `--project-directory` setting, the project name changes to `docker` because that is the name of the directory containing the `src/main/docker/docker-compose.yml` file. By adding a `-p` option, we can set the project name to anything we want.

274.4. Start Services in Background

We can start the processes in the background by adding the `-d` option.

```
$ export HOST_API_PORT=1234 && docker-compose -f src/main/docker/docker-compose.yml up
-d ①
[+] Running 3/3
✓ Container docker-mongodb-1 Started
✓ Container docker-postgres-1 Started
✓ Container docker-api-1     Started
$ ①
```

① `-d` option starts all services in the background and returns us to our shell prompt

274.5. Access Service Logs

With the services running in the background, we can access the logs using the `docker-compose logs` command.

```
$ docker-compose -f src/main/docker/docker-compose.yml logs api ①
```

```
$ docker-compose -f src/main/docker/docker-compose.yml logs -f api mongo ②  
$ docker-compose -f src/main/docker/docker-compose.yml logs --tail 10 ③
```

- ① returns all logs for the `api` service
 - ② tails the current logs for the `api` and `mongo` services.
 - ③ returns the latest 10 messages in each log

274.6. Stop Running Services

If the services were started in the foreground, we can simply stop them with the `<ctl>+C` command. If they were started in the background or in a separate shell, we can stop them by executing the `down` command in the `docker-compose.yml` directory.

```
$ docker-compose -f src/main/docker/docker-compose.yml down
[+] Running 4/4
✓ Container docker-api-1      Removed
✓ Container docker-postgres-1 Removed
✓ Container docker-mongodb-1  Removed
✓ Network docker default     Removed
```

Chapter 275. Docker Cleanup

Docker Compose will mostly cleanup after itself. The only exceptions are the older versions of the API image and the builder image that went into creating the final API images. Using my example settings, these are all end up being named and tagged as `none` in the images repository.

Example Docker Image Repository State

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
docker-hello-example	layered	9c45ff5ac1cf	17 hours ago	316MB
registry.heroku.com/ejava-docker/web	latest	9c45ff5ac1cf	17 hours ago	316MB
docker-hello-example	execjar	669de355e620	46 hours ago	315MB
dockercompose-votes-api	latest	da94f637c3f4	5 days ago	340MB
<none>	<none>	d64b4b57e27d	5 days ago	397MB
<none>	<none>	c5aa926e7423	7 days ago	340MB
<none>	<none>	87e7aab6049	7 days ago	397MB
<none>	<none>	478ea5b821b5	10 days ago	340MB
<none>	<none>	e1a5add0b963	10 days ago	397MB
<none>	<none>	4e68464bb63b	11 days ago	340MB
<none>	<none>	b09b4a95a686	11 days ago	397MB
...				
<none>	<none>	ee27d8f79886	4 months ago	396MB
adoptopenjdk	14-jre-hotspot	157bb71cd724	5 months ago	283MB
mongo	4.4.0-bionic	409c3f937574	12 months ago	493MB
postgres	12.3-alpine	17150f4321a3	14 months ago	157MB
<none>	<none>	b08caee4cd1b	41 years ago	279MB
docker-hello-example	6.1.0-SNAPSHOT	a855dabfe552	41 years ago	279MB

Docker Images are Actually Smaller than Provided SIZE



Even though Docker displays each of these images as >300MB, they may share some base layers and—by themselves—much smaller. The value presented is the

space taken up if all other images are removed or if this image was exported to its own TAR file.

275.1. Docker Image Prune

The following command will clear out any docker images that are not named/tagged and not part of another image.

Example Docker Image Prune Output

```
$ docker image prune
WARNING! This will remove all dangling images.
Are you sure you want to continue? [y/N] y
Deleted Images:
deleted: sha256:ebc8dcf8cec15db809f4389efce84afc1f49b33cd77cf19066a1da35f4e1b34
...
deleted: sha256:e4af263912d468386f3a46538745bfe1d66d698136c33e5d5f773e35d7f05d48

Total reclaimed space: 664.8MB
```

275.2. Docker System Prune

The following command performs the same type of cleanup as the `image` prune command and performs an additional amount on cleanup many other Docker areas deemed to be "trash".

Example Docker System Prune Output

```
$ docker system prune
WARNING! This will remove:
 - all stopped containers
 - all networks not used by at least one container
 - all dangling images
 - all dangling build cache

Are you sure you want to continue? [y/N] y
Deleted Networks:
testcontainers-votes-spock-it_default

Deleted Images:
deleted: sha256:e035b45628fe431901b2b84e2b80ae06f5603d5f531a03ae6abd044768eec6cf
...
deleted: sha256:c7560d6b795df126ac2ea532a0cc2bad92045e73d1a151c2369345f9cd0a285f

Total reclaimed space: 443.3MB
```

You can also add `--volumes` to cleanup orphan volumes as well.

```
$ docker system prune --volumes
```

275.3. Image Repository State After Pruning

After pruning the images — we have just the named/tagged image(s).

Docker Image Repository State After Pruning

\$ docker images	TAG	IMAGE ID	CREATED
REPOSITORY			
SIZE			
docker-hello-example	layered	9c45ff5ac1cf	17 hours ago
316MB			
registry.heroku.com/ejava-docker/web	latest	9c45ff5ac1cf	17 hours ago
316MB			
docker-hello-example	execjar	669de355e620	46 hours ago
315MB			
mongo	4.4.0-bionic	409c3f937574	12 months ago
493MB			
postgres	12.3-alpine	17150f4321a3	14 months ago
157MB			
docker-hello-example	6.1.0-SNAPSHOT	a855dabfe552	41 years ago
279MB			

Chapter 276. Summary

In this module, we learned:

- the purpose of Docker Compose and how it is used to define a network of services operating within a virtualized Docker environment
- to create a Docker Compose file that defines a network of services and their dependencies
- to custom configure a Docker Compose network for different uses
- perform Docker Compose lifecycle commands
- execute ad-hoc commands inside running images

Why We Covered Docker and Docker Compose



The Docker and Docker Compose lectures have been included in this course because of the high probability of your future deployment environments for your Web applications and to provide a more capable and easy to use environment to learn, develop, and debug.

Where are You?



This lecture leaves you at a point where your Web application and database instances are alive but not yet communicating. However, we have much to do before then.

Where are You Going?



In the near future we will dive into the persistence tier, do some local development with the resources we have just setup, and then return to this topic once we are ready to re-deploy with a database-ready Web application.

Docker Compose Integration Testing

copyright © 2024 jim stafford (jim.stafford@jhu.edu)

Chapter 277. Introduction

In the last lecture, we looked at a set of containerized services that we could conveniently launch and manage with Docker Compose.

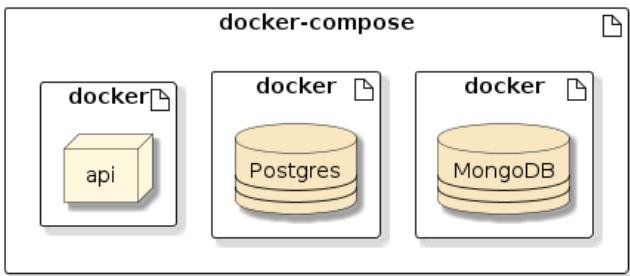


Figure 126. Our Services can be Managed using Docker Compose

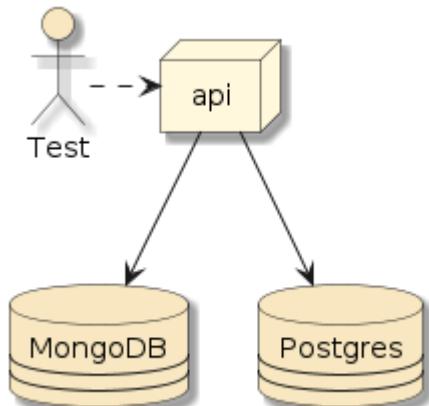


Figure 127. We want to Test API using Close-to-Real Services

We could use in-memory/fakes, but that would not take advantage of how close we could get our tests to represent reality using Docker.



Technically, Postgres and MongoDB do not have in-memory options. Postgres can be simulated with an SQL-compliant H2 database. There are some test fakes for MongoDB, but they are highly version and capability constrained.

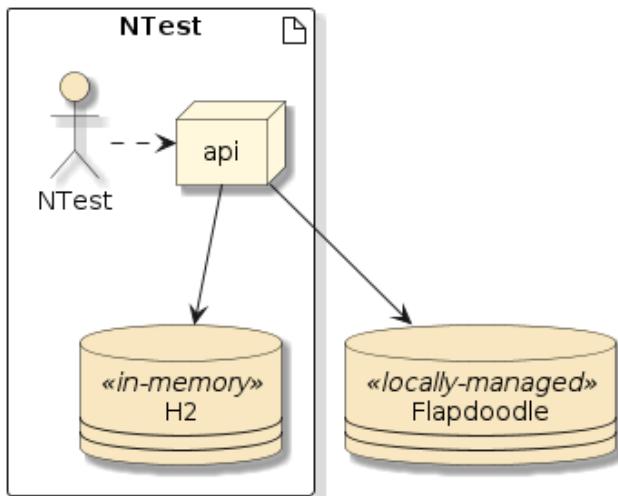


Figure 128. Testing with In-Memory/Local Resources

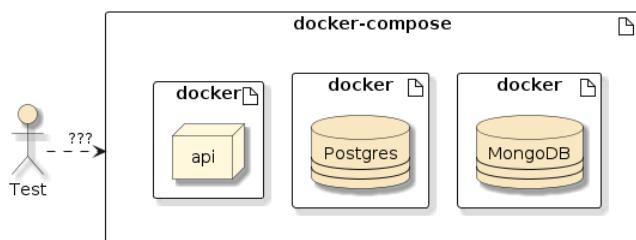
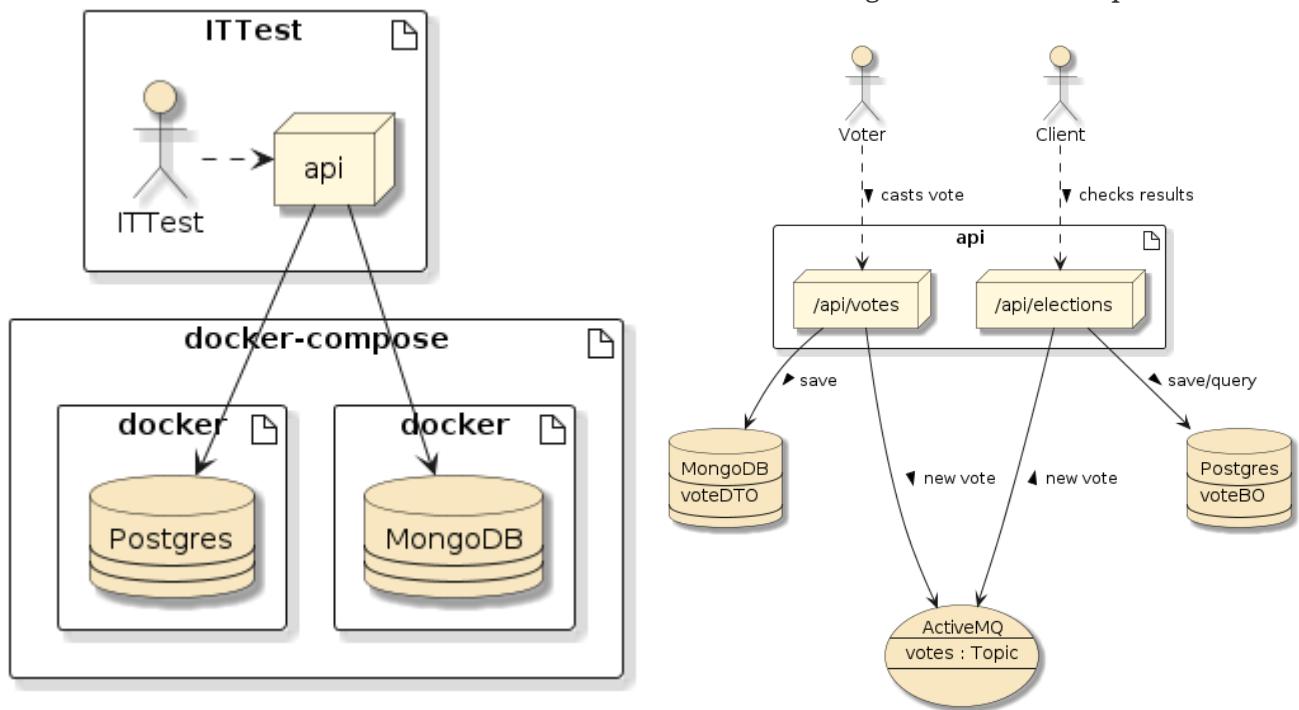


Figure 129. How Can We Test with Real Resources

What if we needed to test with a real or specific version of MongoDB, Postgres, or some other resource? We want to somehow integrate our development and testing with our capability to manage containers with Docker Compose and manage test lifecycles with Maven.

We want to develop and debug within the IDE and still use containerized dependencies.

We want to potentially package our API into a Docker image and then test it in semi-runtime conditions using containerized dependencies.



In this lecture, we will explore using Docker Compose as a development and Failsafe Integration Test aid.

277.1. Goals

You will learn:

- how to automate a full end-to-end heavyweight Failsafe integration test with Maven and Docker Compose
- how to again resolve networking issues when running in a CI/CD environment
- how to address environment-specific JVM setup

277.2. Objectives

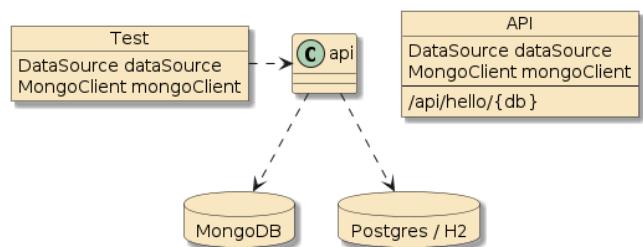
At the conclusion of this lecture and related exercises, you will be able to:

1. automate an end-to-end Failsafe integration test with Maven plugins and Docker Compose
2. setup host specifications relative to the host testing environment
3. insert a custom wrapper script to address environment-specific JVM processing options

Chapter 278. Starting Point

I will start this Docker Compose Integration Test lecture with an example set of tests and application that simply establishes a connection to the services. We will cover more about database and connection options in the next several weeks during the persistence topics, but right now we want to keep the topic focused on setting up the virtual environment and less about the use of the databases specifically.

The examples will use an API controller and tests injected with an RDBMS ([DataSource](#)) and MongoDB ([MongoClient](#)) components to verify the end-to-end communication is in place.



We will start with a Unit Integration Test (NTest) that configures the components with an in-memory H2 RDBMS—which is a common practice for this type of test.

- Postgres does not provide an embedded/in-memory option. Postgres and H2 are both SQL-compliant relational databases, and H2 will work fine for this level of test.
- MongoDB does not provide an embedded/in-memory option. I will cover some "embedded" MongoDB test options in the future persistence lectures. Until then, I will skip over the MongoDB testing option until we reach the Docker-based Maven Failsafe sections.

The real Postgres and MongoDB instances will be part of the follow-on sections of this lecture when we bring in Docker Compose.

278.1. Maven Dependencies

We add [starter-data-jpa](#) and [starter-data-mongo](#) dependencies to the module that will bootstrap the RDBMS and MongoDB database connections. These dependencies are added with [compile](#) scope since this will be directly used by our deployed production code and not limited to test.

For RDBMS, we will need a Postgres dependency for test/runtime and an H2 dependency for test. There is no [src/main](#) compile dependency on Postgres or H2. The additional dependencies supply DB-specific implementations of relational interfaces that our [src/main](#) code does not need to directly depend upon.

Required Maven Dependencies

```
<dependency> ①
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-mongodb</artifactId>
</dependency>
```

```

<dependency> ②
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>test</scope>
</dependency>

```

① MongoDB and RDBMS interfaces our implementation code must depend upon

② RDBMS implementation code needed at test time or operational runtime

278.2. Unit Integration Test Setup

The H2 In-memory Unit Integration Test shows an injected `javax.sql.DataSource` that will be supplied by the JPA AutoConfiguration. There will be a similar MongoDB construct once we start using external databases.

H2 In-memory NTest

```

import javax.sql.DataSource;

@SpringBootTest(classes={DockerComposeHelloApp.class,
    ClientTestConfiguration.class},
    webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
@ActiveProfiles({"test","inmemory"})
class HelloH2InMemoryNTest {
    @Value("${spring.datasource.url}")
    private String expectedJdbcUrl;
    @Autowired
    private DataSource dataSource;

```

The in-memory properties file supplies an H2 database that is suitable for running basic relational tests. We will inject that value directly into the test and have the `starter-data-jpa` module use that same URL to build the `DataSource`.

inmemory Profile Properties File

```

#DB to use during Unit Integration tests
spring.datasource.url=jdbc:h2:mem:dockercompose

```

278.3. Unit Integration Test

All we want to do during this example is to create a connection to the RDBMS database and verify that we have the expected connection. The snippet below shows a test that:

- verifies a DataSource was injected
- verifies the URL for connections obtained match expected

```
@Test
void can_get_connection() throws SQLException {
    //given
    then(dataSource.isNotNull()); ①
    String jdbcUrl;
    //when
    try(Connection conn= dataSource.getConnection()) { ②
        jdbcUrl=conn.getMetaData().getURL();
    }
    //then
    then(jdbcUrl).isEqualTo(expectedJdbcUrl); ③
    then(jdbcUrl).isEqualTo(expectedJdbcUrl);
    then(jdbcUrl).contains("jdbc:h2:mem"); ④
}
```

- ① container established `javax.sql.DataSource` starting point for SQL connections
 ② establish connection to obtain URL using Java's *try-with-resources* AutoCloseable feature for the connection
 ③ connection URL should match injected URL from property file
 ④ URL will start with standard `jdbc` URL for H2 in-memory DB

278.4. DataSource Service Injection

The example contains a very simple Spring MVC controller that will return the database URL for the injected `DataSource`.

Demo Controller to show Runtime Injection

```
@RestController
@RequiredArgsConstructor
public class HelloDBController {
    private final DataSource dataSource;

    @GetMapping(path="/api/hello/jdbc",
            produces = {MediaType.TEXT_PLAIN_VALUE})
    public String helloDataSource() throws SQLException {
        try (Connection conn = dataSource.getConnection()) {
            return conn.getMetaData().getURL();
        }
    }
    ...
}
```

278.5. DataSource Service Injection Test

The test invokes the controller endpoint and verifies the database URL is what is expected.

Test to Verify DB URL used by API Service

```
@Test
void server_can_get_jdbc_connection() {
    //given
    URI url = helloDBUrl.build("jdbc");
    RequestEntity<Void> request = RequestEntity.get(url).build();
    //when
    ResponseEntity<String> response = anonymousUser.exchange(request, String.class);
    //then
    then(response.getStatusCode()).isEqualTo(HttpStatus.OK);
    String jdbcUrl=response.getBody();
    then(jdbcUrl).isEqualTo(expectedJdbcUrl); ①
    then(jdbcUrl).contains("jdbc:h2:mem"); ②
}
```

① database URL should match URL injected from property file

② database URL should be a standard `jdbc` URL for an in-memory H2 database

278.6. DataSource Service Injection Test Result

The following snippet shows a portion of the in-memory RDBMS test results. The snippet below shows the server-side endpoint returning the JDBC URL used to establish a connection to the H2 database.

DataSource Service Injection Test Result

```
GET /api/hello/jdbc, headers=[accept:"text/plain, application/json, application/xml,
text/xml, application/*+json, application/*+xml, */*", user-agent:"masked",
host:"localhost:54539", connection:"keep-alive"]]

rcvd: [Content-Type:"text/plain;charset=UTF-8", Content-Length:"25", Date:"Tue, 06 Feb
2024 17:22:17 GMT", Keep-Alive:"timeout=60", Connection:"keep-alive"]
jdbc:h2:mem:dockercompose
```

At this point we can do development with the injected `DataSource` and work primarily within the IDE (no IT lifecycle test). A similar setup will be shown for MongoDB when we add Docker Compose. Let's start working on the Docker Compose aspects in the next section.

Chapter 279. Docker Compose Database Services

We will first cover what is required to use the database services during development. Our sole intention is to be able to connect to a live Postgres and MongoDB from JUnit as well as the API component(s) under test.

279.1. Docker Compose File

The snippet below shows a familiar set of Postgres and MongoDB database services from the previous Docker Compose lecture. For this lecture, I added the `ports` definition so that we could access the databases from our development environment in addition to later in the test network.

Both databases will be exposed to 0.0.0.0:port# on the Docker Host (which likely will be localhost during development). Postgres will be exposed using port 5432 and MongoDB using port 27017 on the Docker Host. We will have the opportunity to assign random available port numbers for use in automated Unit Integration Tests to keep tests from colliding.

```
services:  
  postgres:  
    image: postgres:12.3-alpine  
    environment:  
      POSTGRES_PASSWORD: secret  
    ports: #only needed when IT tests are directly using DB  
      - "${HOST_POSTGRES_PORT:-5432}:5432" ① ③  
  mongodb:  
    image: mongo:4.4.0-bionic  
    environment:  
      MONGO_INITDB_ROOT_USERNAME: admin  
      MONGO_INITDB_ROOT_PASSWORD: secret  
    ports: #only needed when IT tests are directly using DB  
      - "${HOST_MONGO_PORT:-27017}:27017" ② ③
```

① Postgres will be available on the Docker Host: `HOST_POSTGRES_PORT` (possibly localhost:5432 by default)

② MongoDB will be available on the Docker Host: `HOST_MONGO_PORT` (possibly localhost:27017 by default)

③ if only port# is specified, the port# will listen on all network connections (expressed as `0.0.0.0`)

279.2. Manually Starting Database Containers

With that definition in place, we can launch our two database containers exposed as 0.0.0.0:27017 and 0.0.0.0:5432 on the localhost. `0.0.0.0` is a value that means "all network interfaces"—internal-only (localhost) and externally exposed.

```
$ docker-compose -f src/main/docker/docker-compose.yml up mongodb postgres
...
[+] Running 2/0
  ✓ Container docker-postgres-1 Created      0.0s
  ✓ Container docker-mongodb-1 Created      0.0s
Attaching to mongodb-1, postgres-1
...
-- 
$ docker ps
CONTAINER ID   IMAGE          PORTS          NAMES
9856be1d8504   mongo:4.4.0-bionic   0.0.0.0:27017->27017/tcp   docker-mongodb-1
47d8346e2a78   postgres:12.3-alpine  0.0.0.0:5432->5432/tcp   docker-postgres-1
```

279.3. Postgres Container IT Test

Knowing we will be able to have a live set of databases at test time, we can safely author an IT test that looks very similar to the earlier NTest except that it uses 2 new profile property files.

- `containerdb` - is a profile that sets the base properties for remote Postgres (versus in-memory H2) and MongoDB instances. The host and port numbers will be dynamically assigned during that time by Maven plugins and a property filter.
- `containerdb-dev` - is a profile that overrides test-time settings during development. The host and port numbers will be fixed values.

```
@SpringBootTest(classes={DockerComposeHelloApp.class,
    ClientTestConfiguration.class},
    webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
//uncomment containerdb-dev when developing/running against a fixed instance
@ActiveProfiles({"test", "containerdb", "containerdb-dev"})
@Slf4j
class HelloPostgresContainerIT {
    @Value("${spring.datasource.url}")
    private String expectedJdbcUrl;
    @Autowired
    private DataSource dataSource;
```

@ActiveProfile can be programmatically defined



We will cover the use of an `ActiveProfilesResolver` capability later in the course within the MongoDB section—that will allow the `@ActiveProfile` to be programmatically defined without source code change.

The primary `containerdb` property file is always active and will reference the Postgres and MongoDB databases using \${placeholders} for the Docker Host and port numbers.

application-containerdb.properties

```
#used when running local application Test against remote DB
spring.datasource.url=jdbc:postgresql://${ejava-
parent.docker.hostname}:${host.postgres.port}/postgres ①
spring.datasource.username=postgres
spring.datasource.password=secret

spring.data.mongodb.uri=mongodb://admin:secret@${ejava-
parent.docker.hostname}:${host.mongodb.port}/test?authSource=admin ①
```

① placeholders will be replaced by literal values in pom when project builds

The **containerdb-dev** property file removes the \${placeholders} and uses localhost:fixed-port# to reference the running Postgres and MongoDB databases during development. This must get commented out before running the automated tests using Maven and Failsafe during the build.

application-containerdb-dev.properties

```
#used when running local application Test against remote DB
spring.datasource.url=jdbc:postgresql://localhost:5432/postgres ①
spring.data.mongodb.uri=mongodb://admin:secret@localhost:27017/test?authSource=admin
①
```

① this assumes Docker Host is **localhost** during active IDE development

The net result of the layered set of property files is that the Postgres username and password will be re-used while the URLs are being wholesale replaced.



Placeholders are Replaced Before Runtime

Since we are using Maven resource filtering, the application-containerdb.properties placeholders will be replaced during the build. That means the placeholders will contain literal values during test execution and cannot be individually overridden.

279.4. Postgres Container IT Result

The following snippet shows the example results of running the IT test within the IDE, with the "-dev" profile activated, and running against the Postgres database container manually started using Docker Compose.

```
GET http://localhost:54204/api/hello/jdbc, returned 200 OK/200
sent: [Accept:"text/plain, application/json, application/xml, text/xml,
application/*+json, application/*+xml, */*, Content-Length:"0"]

rcvd: [Content-Type:"text/plain; charset=UTF-8", Content-Length:"41", Date:"Tue, 06 Feb
2024 16:58:58 GMT", Keep-Alive:"timeout=60", Connection:"keep-alive"]
jdbc:postgresql://localhost:5432/postgres ①
```

① last line comes from `conn.getMetaData().getURL()` in the controller

From here, we can develop whatever RDBMS code we need to develop, set breakpoints, write/execute tests, etc. before we commit/push for automated testing.

279.5. MongoDB Container IT Test

We have a similar job to complete with MongoDB. Much will look familiar. In this case, we will inject the source database URL and a `MongoClient` that has been initialized with that same database URL.

MongoDB Container IT Test

```
import com.mongodb.client.MongoClient;  
...  
@SpringBootTest(classes={DockerComposeHelloApp.class,  
    ClientTestConfiguration.class},  
    webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)  
//uncomment containerdb-dev when developing/running against a fixed instance  
@ActiveProfiles({"test", "containerdb", "containerdb-dev"})  
class HelloMongoContainerIT {  
    @Value("${spring.data.mongodb.uri}")  
    private String expectedMongoUrl;  
    @Autowired  
    private MongoClient mongoClient;
```

We will verify the `MongoClient` is using the intended URL information. Unfortunately, that is not a simple `getter()`—so we need to do some regular expression evaluations within a block of cluster description text.

MongoClient Injection Test

```
@Test  
void can_get_connection() throws SQLException, JsonProcessingException {  
    //given  
    then(mongoClient).isNotNull();  
    //when  
    String shortDescription = mongoClient.getClusterDescription().getShortDescription()  
    ();  
    //then  
    new MongoVerifyTest().actual_hostport_matches_expected(expectedMongoUrl,  
    shortDescription);  
}
```

The following snippet shows two (2) example cluster `shortDescription` Strings that can be returned.

Example Cluster Short Descriptions

```
{type=STANDALONE, servers=[{address=localhost:56295, type=STANDALONE,
```

```
roundTripTime=20.3 ms, state=CONNECTED}]} ①
```

```
{type=STANDALONE, servers=[{address=host.docker.internal:56295, type=STANDALONE,  
roundTripTime=20.3 ms, state=CONNECTED}]} ②
```

① example expected shortDescription() when Docker Host is localhost

② example expected shortDescription() during automated testing within CI/CD Docker container when localhost is not the Docker Host

The following snippet shows some of the details to extract the host:portNumber from the shortDescription and mongoDB URL to see if they match.

MongoDB URL Validation Utility

```
public class MongoDBVerifyTest {  
    //..., servers=[{address=localhost:56295, type=STANDALONE... ①  
    private static final Pattern DESCR_ADDRESS_PATTERN = Pattern.compile("address=(A-  
Za-z\\.:0-9]+);");  
    //mongodb://admin:secret@localhost:27017/test?authSource=admin ②  
    private static final Pattern URL_HOSTPORT_PATTERN = Pattern.compile("@([A-Za-z  
\\.:0-9]+)/");  
  
    void actual_hostport_matches_expected(String expectedMongoUrl, String description)  
{  
    Matcher m1 = DESCR_ADDRESS_PATTERN.matcher(description);  
    then(expectedMongoUrl).matches(url->m1.find(), DESCR_ADDRESS_PATTERN.toString()  
());  
  
    Matcher m2 = URL_HOSTPORT_PATTERN.matcher(expectedMongoUrl);  
    then(expectedMongoUrl).matches(url->m2.find(), URL_HOSTPORT_PATTERN.toString()  
());  
  
    then(m1.group(1)).isEqualTo(m2.group(1));  
}
```

① m1 is targeting the runtime result

② m2 is targeting the configuration property setting

279.6. MongoClient Service Injection

The following snippet shows a server-side component that has been injected with the MongoClient and will return the cluster short description when called.

MongoClient Service Injection

```
@RestController  
 @RequiredArgsConstructor  
 @Slf4j  
 public class HelloDBController {
```

```

private final MongoClient mongoClient;
...
    @GetMapping(path="/api/hello/mongo",
        produces = {MediaType.TEXT_PLAIN_VALUE})
    public String helloMongoClient() throws SQLException {
        return mongoClient.getClusterDescription().getShortDescription();
    }
}

```

The following snippet shows an API test—very similar to the Postgres "container DB" test—that verifies we have MongoDB client injected on the server-side and will be able to communicate with the intended DB server when needed.

MongoClient Service Injection Test

```

@Test
void server_can_get_mongo_connection() {
    //given
    URI url = helloDBUrl.build("mongo");
    RequestEntity<Void> request = RequestEntity.get(url).build();
    //when
    String shortDescription = anonymousUser.exchange(request, String.class).getBody();
    //then
    new MongoVerifyTest().actual_hostport_matches_expected(expectedMongoUrl,
    shortDescription);
}

```

279.7. MongoDB Container IT Result

The following snippet shows an exchange between the IT test client and the server-side components. The endpoint returns the cluster short description text with the host:portNumber value from the injected `MongoClient`.

MongoDB Container IT Result

```

GET http://localhost:54507/api/hello/mongo, returned 200 OK/200
sent: [Accept:"text/plain, application/json, application/xml, text/xml,
application/*+json, application/*+xml, */*, Content-Length:"0"]

rcvd: [Content-Type:"text/plain;charset=UTF-8", Content-Length:"110", Date:"Tue, 06
Feb 2024 17:19:09 GMT", Keep-Alive:"timeout=60", Connection:"keep-alive"]
{type=STANDALONE, servers=[{address=localhost:27017, type=STANDALONE,
roundTripTime=70.2 ms, state=CONNECTED}] ①

```

① last line comes from `mongoClient.getClusterDescription().getShortDescription()` call in the controller

Chapter 280. Automating Container DB IT Tests

We have reached a point where:

- the server-side is injected with a database connection,
- the integration tests inquire about that connection, and
- the test results confirm whether the connection injected into the server has the correct properties

We can do that within the IDE using the `-dev` profile. To complete the module, we need wire the test to be automated within the Maven build.

280.1. Generate Random Available Port Numbers

We already have the `build-helper-maven-plugin` automatically activated because of the `application-it.properties` activation file being in place. The default plugin definition from the `ejava-build-parent` will generate a random available port# for `server.http.port`. We will continue to use that for the API and now generate two additional port numbers for database communications.

The following snippet shows an extension of the parent plugin declaration to include ports to expose Postgres and MongoDB on the Docker Host.

Generate Random Database Ports

```
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>build-helper-maven-plugin</artifactId>
    <executions>
        <execution>
            <id>reserve-network-port</id>
            <configuration>
                <portNames combine.children="append"> ①
                    <portName>host.postgres.port</portName>
                    <portName>host.mongodb.port</portName>
                </portNames>
            </configuration>
        </execution>
    </executions>
</plugin>
```

① two (2) additional `portName`s will be generated during the parent-defined `process-resources` phase



There are at least two ways to extend a parent definition for collections like `portNames`:

- `combine.children="append"` — adds child elements to parent elements
- `combine.parent="override"` — replaces the parent elements with child elements

The following snippet shows the impact of the above definition when executing the [process-resources](#) page.

Port Allocation Example Output

```
$ mvn process-resources
...
[INFO] --- build-helper-maven-plugin:3.4.0:reserve-network-port (reserve-network-port)
@ dockercompose-it-example ---
[INFO] Reserved port 55225 for server.http.port
[INFO] Reserved port 55226 for host.postgres.port
[INFO] Reserved port 55227 for host.mongodb.port
```

We will use these variables to impact the containers and integration tests.

280.2. Launch Docker Compose with Variables

With the port number variables defined, we can start the network of containers using Docker Compose. The following snippet shows using the [io.brachu:docker-compose-maven-plugin](#) to implement the execution.

Launch Docker Compose

```
<plugin>
  <groupId>io.brachu</groupId>
  <artifactId>docker-compose-maven-plugin</artifactId>
  <configuration>
    <projectName>${project.artifactId}</projectName> ①
    <workDir>${project.basedir}/src/main/docker</workDir> ②
    <file>${project.basedir}/src/main/docker/docker-compose.yml</file> ③
    <skip>${skipITs}</skip> ④
    <env> ⑤
      <HOST_API_PORT>${server.http.port}</HOST_API_PORT>
      <HOST_POSTGRES_PORT>${host.postgres.port}</HOST_POSTGRES_PORT>
      <HOST_MONGODB_PORT>${host.mongodb.port}</HOST_MONGODB_PORT>
    </env>
    <wait> ⑥
      <value>${it.up.timeout.secs}</value>
      <unit>SECONDS</unit>
    </wait>
    <forceBuild>true</forceBuild> ⑦
    <removeOrphans>true</removeOrphans> ⑧
  </configuration>
  <executions>
    <execution>
```

```

<goals> ⑨
  <goal>up</goal>
  <goal>down</goal>
</goals>
</execution>
</executions>
</executions>
</plugin>

```

- ① project name identifies a common alias each of the services are scoped under
- ② directory to resolve relative paths in `docker-compose.yml` file
- ③ we have placed the `docker-compose.yml` file in a sub-directory
- ④ we do not want to run this plugin if `install` is invoked with `-DskipITs`
- ⑤ these variables are expanded by Maven and passed into Docker Compose as environment variables
- ⑥ maximum time to wait for container to successfully start before failing
- ⑦ for source images, will force a build before running (e.g., run `--build`)
- ⑧ containers will be removed after being stopped (e.g., run `--rm`)
- ⑨ `up` already assigned to `pre-integration-test` and `down` assigned to `post-integration-test`

There are several uses of a directory context at play. They each are used to resolve relative path references.

- Docker build making reference to files copied
- Docker Compose build command making reference to the `Dockerfile`

The simplest setup is when the `Dockerfile` and `docker-compose.yml` file(s) are at the root of the module. In that case, the context defaults to `${project.basedir}` and all module artifacts are in sight.



If the `docker-compose.yml` and `Dockerfile` are placed within the `src` tree (e.g., `src/main/docker`), docker-compose will default the context to the `docker-compose.yml` directory. That can lead to some confusing relative references.

This example tries to make the command line simple, while still automating by:

- setting the `docker-compose-maven-plugin workDir` property to the directory of the `docker-compose.yml` file.
- sets `docker-compose.yml context` property to `../../..` (a.k.a. `${project.basedir}`)
- has all other file references relative to `${project.basedir}`

280.3. Filtering Test Resources

We also need to filter the test property files with our generated Maven properties:

- `ejava-parent.docker.hostname`—hostname of the Docker Host running images. This is being determined in the `ejava-build-parent` parent pom.xml.
- `host.postgres.port`—Docker Host port number to access Postgres
- `host.mongodb.port`—Docker Host port number to access MongoDB

The following snippet shows the source version of our property file—containing placeholders that match the generated Maven properties

`src/test/resources/application-containerdb.properties`

```
spring.datasource.url=jdbc:postgresql://${ejava-
parent.docker.hostname}:${host.postgres.port}/postgres ① ②
spring.datasource.username=postgres
spring.datasource.password=secret

spring.data.mongodb.uri=mongodb://admin:secret@${ejava-
parent.docker.hostname}:${host.mongodb.port}/test?authSource=admin ① ③
```

① `${ejava-parent.docker.hostname}` is being defined in `ejava-build-parent` pom.xml

② `${host.postgres.port}` is being defined in this project's pom.xml using `build-helper-maven-plugin`

③ `${host.mongodb.port}` is being defined in this project's pom.xml using `build-helper-maven-plugin`

These are being defined in a file. Therefore we need to update the file's values at test time. Maven provides first-class support for filtering placeholders in resources files. So much so, that one can define the filtering rules outside the scope of the resources plugin. The following (overly verbose) snippet below identifies two `testResource` actions—one with and one without filtering:

- `filtering=true`—`containerdb.properties` will be expanded when copied
- `filtering=false`—other resource files will be copied without filtering

Test Resource Filtering

```
<build>
  <testResources>
    <testResource>
      <directory>src/test/resources</directory>
      <filtering>true</filtering> ①
      <includes>
        <include>application-containerdb.properties</include>
      </includes>
    </testResource>
    <testResource>
      <directory>src/test/resources</directory>
      <filtering>false</filtering> ②
      <excludes>
        <exclude>application-containerdb.properties</exclude>
      </excludes>
    </testResource>
```

```
</testResources>
```

- ① all resources matching this specification will have placeholders subject to filtering
- ② all resources subject to this specification will be copied as provided

Target Specific Files when Filtering



When adding resource filtering, it is good to identify specific files to filter versus all files. Doing it that way helps avoid unexpected and/or unwanted expansion to other resource files.

Resource File Copied to target Tree is modified



Maven is copying the resource file(s) from the `src/test/resources` tree to `target/test-classes`. `target/test-classes` will be part of the test classpath. Our configuration is adding an optional filtering to the copy process. This does not change the original in `src/test/resources`.



You cannot just specify the `filter=true` specification. If you do—the files that do not match the specification will not be copied at all.

The following snippet shows the result of processing the test resource files. One file was copied (filtered=true) followed by five (5) files (filtered=false). The test resource files are being copied to the `target/test-classes` directory.

Resource Filtering Output

```
$ mvn process-test-resources
...
[INFO] --- build-helper-maven-plugin:3.4.0:reserve-network-port (reserve-network-port)
@ dockercompose-it-example ---
[INFO] Reserved port 56026 for server.http.port
[INFO] Reserved port 56027 for host.postgres.port
[INFO] Reserved port 56028 for host.mongodb.port
...
[INFO] --- maven-resources-plugin:3.3.1:testResources (default-testResources) @
dockercompose-it-example ---
[INFO] Copying 1 resource from src/test/resources to target/test-classes ①
[INFO] Copying 5 resources from src/test/resources to target/test-classes ②
```

- ① 1 file copied with property filtering applied
- ② remaining files copied without property filtering

The snippet below shows the output result of the filtering. The `target/test-classes` copy of the file has the known placeholders expanded to their build-time values.

Resource Filtering Result

```
$ cat target/test-classes/application-containerdb.properties
...
```

```
spring.datasource.url=jdbc:postgresql://localhost:56027/postgres ①
spring.datasource.username=postgres
spring.datasource.password=secret

spring.data.mongodb.uri=mongodb://admin:secret@localhost:56028d/test?authSource=admin
②
```

① \${ejava-parent.docker.hostname} was expanded to `localhost` and \${host.postgres.port} was expanded to `56027`

② \${ejava-parent.docker.hostname} was expanded to `localhost` and \${host.mongodb.port} was expanded to `56028`

280.4. Failsafe

There are no failsafe changes required. All dynamic changes so far are being handled by the test property filtering and Docker Compose.

280.5. Test Execution

We start out test with a `mvn clean verify` to run all necessary phases. The snippet below shows a high-level view of the overall set of module tests.

Test Execution

```
$ mvn clean verify
...
[INFO] --- build-helper-maven-plugin:3.4.0:reserve-network-port (reserve-network-port)
@ dockercompose-it-example ---
[INFO] Reserved port 56176 for server.http.port
[INFO] Reserved port 56177 for host.postgres.port
[INFO] Reserved port 56178 for host.mongodb.port
...
[INFO] --- maven-resources-plugin:3.3.1:testResources (default-testResources) @
dockercompose-it-example ---
[INFO] Copying 1 resource from src/test/resources to target/test-classes
[INFO] Copying 5 resources from src/test/resources to target/test-classes
...
[INFO] --- maven-surefire-plugin:3.1.2:test (default-test) @ dockercompose-it-example
---
...
① [INFO] --- docker-compose-maven-plugin:1.0.0:up (default) @ dockercompose-it-example
---
...
[INFO] --- maven-failsafe-plugin:3.1.2:integration-test (integration-test) @
dockercompose-it-example ---
...
② [INFO] --- docker-compose-maven-plugin:1.0.0:down (default) @ dockercompose-it-example
---
...
```

```
[INFO] --- maven-failsafe-plugin:3.1.2:verify (verify) @ dockercompose-it-example ---
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 01:55 min
```

① (*Test) Unit/Unit Integration Tests are run here

② (*IT) Integration Tests are run here

280.5.1. Test Phase

During the **test** phase, our tests/API run with the in-memory version of the RDBMS. `SpringBootTest` is responsible for the random available port number without any Maven help.

Test Phase

```
[INFO] Running info.ejava.examples.svc.docker.hello.HelloH2InMemoryNTest
...
GET /api/hello/jdbc, headers=[accept:"text/plain, application/json, application/xml,
text/xml, application/*+json, application/*+xml, */*", user-agent:"masked",
host:"localhost:56181", connection:"keep-alive"]]
13:46:58.767 main DEBUG i.e.e.c.web.RestTemplateLoggingFilter#intercept:37
GET http://localhost:56181/api/hello/jdbc, returned 200 OK/200
...
jdbc:h2:mem:dockercompose
```

280.5.2. Docker Compose Up

The Maven Docker Compose Plugin starts the identified services within the `docker-compose.yml` file during the **pre-integration-test** phase.

Docker Compose Up

```
[INFO] --- docker-compose-maven-plugin:1.0.0:up (default) @ dockercompose-it-example
---
...
Network dockercompose-it-example_default Creating
Network dockercompose-it-example_default Created
Container dockercompose-it-example-mongodb-1 Creating
Container dockercompose-it-example-postgres-1 Creating
Container dockercompose-it-example-postgres-1 Created
Container dockercompose-it-example-mongodb-1 Created
Container dockercompose-it-example-postgres-1 Starting
Container dockercompose-it-example-mongodb-1 Starting
Container dockercompose-it-example-mongodb-1 Started
Container dockercompose-it-example-postgres-1 Started
```

280.5.3. MongoDB Container Test Output

The following snippet shows the output of the automated IT test where the client and API are using generated port numbers to reach the API and MongoDB database. The API is running local to the IT test, but the MongoDB database is running on the Docker Host—which happens to be localhost in this case.

MongoDB Container Test Output

```
[INFO] Running info.ejava.examples.svc.docker.hello.HelloMongoDBContainerIT  
...  
GET http://localhost:56216/api/hello/mongo, returned 200 OK/200  
...  
{type=STANDALONE, servers=[{address=localhost:56178, type=STANDALONE,  
roundTripTime=25.2 ms, state=CONNECTED}]}
```

280.5.4. Postgres Container Test Output

The following snippet shows the output of the automated IT test where the client and API are using generated port numbers to reach the API and Postgres database. The API is running local to the IT test, but the Postgres database is running on the Docker Host—which also happens to be localhost in this case.

Postgres Container Test Output

```
[INFO] Running info.ejava.examples.svc.docker.hello.HelloPostgresContainerIT  
...  
GET http://localhost:56216/api/hello/jdbc, returned 200 OK/200  
...  
jdbc:postgresql://localhost:56177/postgres
```

Chapter 281. CI/CD Test Execution

During CI/CD execution, the `${ejava-parent.docker.hostname}` placeholder gets replaced with `host.docker.internal` and mapped to `host-gateway` inside the container to represent the address of the Docker Host.

The following snippet shows an example of the alternate host mapping. Note that the API is still running local to the IT test—therefore it is still using `localhost` to reach the API.

CI/CD Test Execution

```
$ docker-compose run mvn mvn clean verify
...
[INFO] --- build-helper-maven-plugin:3.4.0:reserve-network-port (reserve-network-port)
@ dockercompose-it-example ---
[INFO] Reserved port 40475 for server.http.port
[INFO] Reserved port 41767 for host.postgres.port
[INFO] Reserved port 35869 for host.mongodb.port
...
[INFO] Running info.ejava.examples.svc.docker.hello.HelloMongoDBContainerIT
...
GET http://localhost:40475/api/hello/mongo, returned 200 OK/200 ①
...
{type=STANDALONE, servers=[{address=host.docker.internal:35869, type=STANDALONE,
roundTripTime=43.5 ms, state=CONNECTED}] ②
...
[INFO] Running info.ejava.examples.svc.docker.hello.HelloPostgresContainerIT
...
GET http://localhost:40475/api/hello/jdbc, returned 200 OK/200 ①
...
jdbc:postgresql://host.docker.internal:41767/postgres ②
```

① API is running local to IT tests, thus they use `localhost` to reach

② Postgres and MongoDB containers are running on Docker Host, thus use `host.docker.internal` to reach

Chapter 282. Adding API Container

At this point in the lecture, we have hit most of the primary learning objectives I wanted to make except one: building a Docker image under test using Docker Compose.

282.1. API in Docker Compose

We can add the API to the Docker Compose file—the same as with Postgres and MongoDB. However, there are at least two differences

1. the API is dependent on the `postgres` and `mongodb` services
2. the API is under development—possibly changing—thus should be built

Therefore, there is a separate `build.dockerfile` property to point to the source of the image and a `depends_on` property to identify the dependency/linkage to the other services.

API in Docker Compose

```
api:  
  build:  
    context: ../../.. ①  
    dockerfile: src/main/docker/Dockerfile ②  
    image: dockercompose-hello-example:latest ③  
    ports:  
      - "${HOST_API_PORT:-8080}:8080"  
    depends_on: ④  
      - postgres  
      - mongodb  
    environment: ⑤  
      - DATABASE_URL=postgres://postgres:secret@postgres:5432/postgres  
      - MONGODB_URI=mongodb://admin:secret@mongodb:27017/test?authSource=admin
```

① directory where all relative paths will be based (`#{project.basedir}`)

② source Dockerfile to build image

③ name and tag for built and executed image

④ services to wait for and to add references in `/etc/hosts`

⑤ environment variables to set at runtime

Adding this service to the Docker Compose file will cause the API image to build and container to start along with the Postgres and MongoDB containers.

282.2. API Dockerfile

The following API Dockerfile should be familiar to you—with the addition of the `run_env.sh` file. It is a two stage Dockerfile where:

1. the elements of the final Docker image are first processed and staged

- the elements are put into place and runtime aspects are addressed

```
FROM eclipse-temurin:17-jre AS builder ①
WORKDIR /builder
COPY src/main/docker/run_env.sh .
ARG JAR_FILE=target/*-bootexec.jar
COPY ${JAR_FILE} application.jar
RUN java -Djarmode=tools -jar application.jar extract --layers --launcher
--destination extracted

FROM eclipse-temurin:17-jre ②
WORKDIR /application
COPY --from=builder /builder/extracted/dependencies/ ./
COPY --from=builder /builder/extracted/spring-boot-loader/ ./
COPY --from=builder /builder/extracted/snapshot-dependencies/ ./
COPY --from=builder /builder/extracted/application/ ./
COPY --chmod=555 --from=builder /builder/run_env.sh ./
#https://github.com/spring-projects/spring-boot/issues/37667
ENTRYPOINT ["./run_env.sh",
"java","org.springframework.boot.loader.JarLauncher"]
```

① process/stage elements of the final image

② final layout/format of elements and runtime aspects of image

282.3. run_env.sh

I have added the run_env.sh to represent a wrapper around the Java executable. It is very common to do this when configuring the JVM execution. In this example, I am simply mapping a single string database URL to standard Spring Data properties.

Environment variables with the following format ...

Compact DB URL Formats

```
DATABASE_URL=postgres://postgres:secret@postgres:5432/postgres
MONGODB_URI=mongodb://admin:secret@mongodb:27017/test?authSource=admin
```

... will be transformed into command line parameters with the following form.

Standard Spring Data Properties

```
--spring.data.datasource.url=jdbc:postgresql://host.docker.internal:5432/postgres
--spring.data.datasource.username=postgres
--spring.data.datasource.password=secret
--spring.data.mongodb.uri=mongodb://admin:secret@mongodb:27017/test?authSource=admin
```

The MongoDB URL is very trivial. Postgres will require a little work with the bash shell.

282.3.1. run_env.sh High Level Skeleton

The following snippet shows the high level skeleton of the `run_env.sh` script we will put in place to address all types of environment variables we will see in our environments. The shell will launch whatever command was passed to it ("\$@") and append the `OPTIONS` that it was able to construct from environment variables. We will place this shell script in the `src/main/docker` directory to be picked up by the Dockerfile.

The resulting script was based upon the much more complicated `example`. I was able to simplify it by assuming the database properties would only be for Postgres and MongoDB.

run_env.sh Environment Variable Script

```
#!/bin/bash

OPTIONS=""

#ref: https://raw.githubusercontent.com/heroku/heroku-buildpack-jvm-
common/main/opt/jdbc.sh
if [[ -n "${DATABASE_URL:-}" ]]; then
    # ...
fi

if [[ -n "${MONGODB_URI:-}" ]]; then
    # ...
fi

exec $@ ${OPTIONS} ①
```

① executing the passed in arguments as the command line, in addition to the contents of `OPTIONS`

282.3.2. Script Output

When complete, a `DATABASE_URL` will be transformed into separate Spring Data JPA `url`, `username`, and `password` properties.

```
$ export DATABASE_URL=postgres://postgres:secret@postgres:5432/postgres && bash
src/main/docker/run_env.sh echo ① ②
--spring.datasource.url=jdbc:postgresql://postgres:5432/postgres
--spring.datasource.username=postgres --spring.datasource.password=secret ③

$ unset DATABASE_URL ④
```

① `DATABASE_URL` environment variable set with compound value

② `echo` command executed with contents of `OPTIONS` containing results of processing `DATABASE_URL` value

③ the contents of `OPTIONS`; output by `echo` command

④ clearing the `DATABASE_URL` property from the current shell environment

A `MONGODB_URI` will be pass thru as a single URL using the Spring Data Mongo `url` property.

```
export MONGODB_URI=mongodb://admin:secret@mongo:27017/test?authSource=admin && bash  
src/main/docker/run_env.sh echo  
--spring.data.mongodb.uri=mongodb://admin:secret@mongo:27017/test?authSource=admin  
  
$ unset MONGODB_URI
```

Running that within Docker, results in the following.

Demonstrating run_env.sh within Docker

```
docker run \  
-e DATABASE_URL=postgres://postgres:secret@postgres:5432/postgres \ ①  
-e MONGODB_URI=mongodb://admin:secret@mongo:27017/test?authSource=admin \  
-v `pwd`/src/main/docker/run_env.sh:/tmp/run_env.sh \ ②  
openjdk:17.0.2 \ ③  
/tmp/run_env.sh echo ④  
  
--spring.datasource.url=jdbc:postgresql://postgres:5432/postgres  
--spring.datasource.username=postgres --spring.datasource.password=secret  
--spring.data.mongodb.uri=mongodb://admin:secret@mongo:27017/test?authSource=admin ⑤
```

① `-e` sets environment variable for running Docker container

② mapping `run_env.sh` in source tree to location within Docker container

③ using a somewhat vanilla Docker image

④ executing `echo` command using `run_env.sh` within Docker container

⑤ output of `run_env.sh` after processing the `-e` environment variables

Lets break down the details of `run_env.sh`.

282.3.3. DataSource Properties

The following script will breakout URL, username, and password using [bash regular expression matching](#) and turn them into Spring Data properties on the command line.

DataSource Properties

```
#DATABASE_URL=postgres://postgres:secret@postgres:5432/postgres  
  
if [[ -n "${DATABASE_URL:-}" ]]; then  
    pattern="^postgres://(.+):(.+)@(.+)$" ①  
    if [[ "${DATABASE_URL}" =~ $pattern ]]; then ②  
        JDBC_DATABASE_USERNAME="${BASH_REMATCH[1]}"  
        JDBC_DATABASE_PASSWORD="${BASH_REMATCH[2]}"  
        JDBC_DATABASE_URL="jdbc:postgresql://${BASH_REMATCH[3]}"
```

```

OPTIONS="${OPTIONS} --spring.datasource.url=${JDBC_DATABASE_URL} "
OPTIONS="${OPTIONS} --spring.datasource.username=${JDBC_DATABASE_USERNAME}"
OPTIONS="${OPTIONS} --spring.datasource.password=${JDBC_DATABASE_PASSWORD}"
else
  OPTIONS="${OPTIONS} --no.match=${DATABASE_URL}" ③
fi
fi

```

- ① regular expression defining three (3) extraction variables: username, password, and URI snippet
- ② if the regular expression finds a match, we will pull that apart and assemble the properties using **BASH_REMATCH**
- ③ if no match is found, **--no.match** is populated with the DATABASE_URL to be printed for debug reasons

282.3.4. MongoDB Properties

The Mongo URL we are using can be passed in as a single URL property. If Postgres was this straight forward, we could have stuck with the **CMD** option.

MongoDB Property

```

if [[ -n "${MONGODB_URI:-}" ]]; then
  OPTIONS="${OPTIONS} --spring.data.mongodb.uri=${MONGODB_URI}"
fi

```

282.4. API Container Integration Test

The API Container IT test is very similar to the others except that it is client-side only. That means there will be no **DataSource** or **MongoClient** to inject and the responses from the API calls should return URLs that reference the **postgres:5432** and **mongodb:27017** host and port numbers—matching the Docker Compose configuration.

API Container IT Test

```

@SpringBootTest(classes=ClientTestConfiguration.class, ①
    webEnvironment = SpringBootTest.WebEnvironment.NONE) ②
@ActiveProfiles({"test", "it"})
//we have project Mongo dependency but don't have or need Mongo for this remote client
@EnableAutoConfiguration(exclude = { ③
    MongoAutoConfiguration.class,
    MongoDataAutoConfiguration.class
})
@Slf4j
class HelloApiContainerIT {
    @Autowired
    private RestTemplate anonymousUser;
}

```

- ① not using `@SpringBootApplication` class
- ② not running a local web server
- ③ helps eliminate meaningless MongoDB errors in the client

I am going to skip over some testing setup details that I am pretty sure you can either guess or find by inspecting the source. The following test contacts the API running within a container within the Docker Compose network and verifies it is using a database connection to a host named `postgres` and using port `5432`.

Postgres DataSource API Test

```
@Test
void server_can_get_jdbc_connection() {
    //given
    String name="jim";
    URI url = helloDBUrl.build("jdbc");
    RequestEntity<Void> request = RequestEntity.get(url).build();
    //when
    String jdbcUrl = anonymousUser.exchange(request, String.class).getBody();
    //then
    then(jdbcUrl).contains("jdbc:postgresql");
}
```

The following test contacts the API as well to verify it is using a database connection to a host named `mongodb` and using port `27017`.

MongoDB Client API Test

```
@Test
void server_can_get_mongo_connection() {
    //given
    URI url = helloDBUrl.build("mongo");
    RequestEntity<Void> request = RequestEntity.get(url).build();
    //when
    String shortDescription = anonymousUser.exchange(request, String.class).getBody();
    //then
    new MongoVerifyTest().actual_hostport_matches_expected(expectedMongoUrl,
    shortDescription);
}
```

282.4.1. Building and Running with API Service

The primary difference in the build—after adding the API service—is the building of the API according to the Dockerfile instructions and then starting all the images during the `pre-integration-test` phase.

Building and Running with API Service

```
[INFO] --- docker-compose-maven-plugin:1.0.0:up (default) @ dockercompose-it-example
```

```
---  
#0 building with "desktop-linux" instance using docker driver  
  
#1 [api internal] load build definition from Dockerfile  
#1 transferring dockerfile: 788B 0.0s done  
#1 DONE 0.0s  
  
#2 [api internal] load metadata for docker.io/library/eclipse-temurin:17-jre  
#2 DONE 0.0s  
  
#3 [api internal] load .dockerignore  
#3 transferring context: 2B done  
#3 DONE 0.0s  
  
#4 [api builder 1/5] FROM docker.io/library/eclipse-temurin:17-jre  
#4 DONE 0.0s  
  
#5 [api internal] load build context  
#5 transferring context: 63.78MB 1.2s done  
#5 DONE 1.2s  
  
#6 [api builder 2/5] WORKDIR /builder  
#6 CACHED  
  
#7 [api builder 3/5] COPY src/main/docker/run_env.sh .  
#7 CACHED  
  
#8 [api builder 4/5] COPY target/*-bootexec.jar application.jar  
#8 DONE 0.6s  
  
#9 [api builder 5/5] RUN java -Djarmode=tools -jar application.jar extract --layers  
--launcher --destination extracted  
#9 DONE 1.6s  
  
#10 [api stage-1 3/7] COPY --from=builder /builder/extracted/dependencies/ ./  
#10 CACHED  
  
#11 [api stage-1 4/7] COPY --from=builder /builder/extracted/spring-boot-loader/ ./  
#11 CACHED  
  
#12 [api stage-1 5/7] COPY --from=builder /builder/extracted/snapshot-dependencies/ ./  
#12 CACHED  
  
#13 [api stage-1 6/7] COPY --from=builder /builder/extracted/application/ ./  
#13 CACHED  
  
#14 [api stage-1 2/7] WORKDIR /application  
#14 CACHED  
  
#15 [api stage-1 7/7] COPY --chmod=555 --from=builder /builder/run_env.sh ./  
#15 CACHED
```

```

#16 [api] exporting to image
#16 exporting layers done
#16 writing image
sha256:cadb45a28f3e8c2d7b521ec4ffda5276e72ccb147225eb1527ee4356f6ba44cf done
#16 naming to docker.io/library/dockercompose-hello-example:latest done
#16 DONE 0.0s

#17 [api] resolving provenance for metadata file
#17 DONE 0.0s
Network dockercompose-it-example_default Creating
Network dockercompose-it-example_default Created
Container dockercompose-it-example-postgres-1 Creating
Container dockercompose-it-example-mongodb-1 Creating
Container dockercompose-it-example-postgres-1 Created
Container dockercompose-it-example-mongodb-1 Created
Container dockercompose-it-example-api-1 Creating
Container dockercompose-it-example-api-1 Created
Container dockercompose-it-example-mongodb-1 Starting
Container dockercompose-it-example-postgres-1 Starting
Container dockercompose-it-example-mongodb-1 Started
Container dockercompose-it-example-postgres-1 Started
Container dockercompose-it-example-api-1 Starting
Container dockercompose-it-example-api-1 Started

```

282.4.2. API Container IT CI/CD Test Output

The following snippet shows key parts of the API Container IT test output when running within the CI/CD environment.

API Container IT CI/CD Test Output

```

[INFO] Running info.ejava.examples.svc.docker.hello.HelloApiContainerIT
...
GET http://host.docker.internal:37875/api/hello/jdbc, returned 200 OK/200 ①
...
jdbc:postgresql://postgres:5432/postgres ②
...
GET http://host.docker.internal:37875/api/hello/mongo, returned 200 OK/200 ①
...
{type=STANDALONE, servers=[{address=mongodb:27017, type=STANDALONE, roundTripTime=79.6
ms, state=CONNECTED}]} ③

```

① API is running in remote Docker image on Docker Host — which is not localhost

② API is using a Postgres database for its DataSource. Postgres is running on a host known to the API as **postgres** because of the Docker Compose file dependency mapping

③ API is using a MongoDB database for its MongoClient. MongoDB is running on a host known to the API as **mongodb** because of the Docker Compose file dependency mapping

Chapter 283. Summary

In this module, we learned:

- to integrate Docker Compose into a Maven integration test phase
- to wrap the JVM execution in a wrapper script to process custom runtime options
- to address development, build, and CI/CD environment impact to hostnames and port numbers.

Testcontainers Unit Integration Testing

copyright © 2024 jim stafford (jim.stafford@jhu.edu)

Chapter 284. Introduction

In previous sections we implemented "unit integration tests" (NTests) and with in-memory instances for back-end resources. This was lightweight, fast, and convenient but lacked the flexibility to work with more realistic implementations.

We later leveraged Docker, Docker Compose, and Maven Failsafe to implement heavyweight "integration tests" with real resources operating in a virtual environment. We self-integrated Docker and Docker Compose using several Maven plugins ([resources](#), [build-helper-maven-plugin](#), [spring-boot-maven-plugin](#), [docker-maven-plugin](#), [docker-compose-maven-plugin](#), [maven-failsafe-plugin](#), ...) and Maven's integration testing phases. It was a mouthful, but it worked.

In this lecture I will introduce an easier, more seamless way to integrate Docker images into our testing using [Testcontainers](#). This will allow us to drop back into the Maven test phase and implement the integration tests using straight forward unit test constructs enabling certain use cases:

- Data access layer implementation testing
- External service integration an testing
- Lightweight development overhead to certain types of integration tests

We won't dive deep into database topics yet. We will continue to just focus on obtaining a database instance and connection.

284.1. Goals

You will learn:

- how to more easily integrate Docker images and Docker Compose into tests
- how to inject dynamically assigned values into the application context startup

284.2. Objectives

At the conclusion of this lecture and related exercises, you will be able to:

1. implement a unit integration test using Docker and Testcontainers—specifically with the [PostgresContainer](#), [MongoDBContainer](#), and [GenericContainer](#)
2. implement a Spring [@DynamicPropertySource](#) to obtain dynamically assigned properties in time for concrete component injections
3. implement a lightweight Maven Failsafe integration test using Docker Compose and Testcontainers—specifically with the [DockerComposeContainer](#)

Chapter 285. Testcontainers Overview

[Testcontainers](#) is a Java library that supports running Docker containers within JUnit tests and other test frameworks. We do not need extra Maven plugins to manage the containers. Testcontainers operates within the JVM and integrates with JUnit.

Maven Plugins Still Needed to Build Internal Artifacts



We still need Maven plugins to build the artifacts (e.g., executable JAR) to package into the Docker image if testing something constructed locally. We just don't need the web of plugins surrounding the [integration-test](#) phases.

Testcontainers provides a layer of integration that is well aware of the integration challenges that are present when testing with Docker images and can work both outside and inside a Docker container itself (i.e., it can operate in local development and CI/CD Docker container environments).

Based on the changes made in Spring Boot 3, Spring and Spring Boot are very enthusiastic about Testcontainers. Spring Boot has also dropped its direct support of Flapdoodle. Flapdoodle is an embedded MongoDB test-time framework also geared for testing like Testcontainers—but specific to MongoDB and constrained to certain versions of MongoDB. By embracing Testcontainers, one is only constrained by what can be packaged into a Docker image.



Flapdoodle can still be integrated with Spring/Spring Boot. However, as we will cover in the MongoDB lectures (and specifically in the [Spring Boot 3 porting notes](#)), the specific dependencies and properties have changed. They are no longer directly associated with Spring/Spring Boot and now come from Flapdoodle sources.

Chapter 286. Base Example

I will start the Testcontainers example with some details using a database connection test that should be familiar to you from earlier testing lectures. This will allow you to see many familiar constructs before we move on to the Testcontainers topics. The goal of the database connection example(s) is to establish a database instance and connection to it. Database interaction will be part of later persistence lectures.

286.1. Maven Dependencies

We will be injecting components that will provide connections to the RDBMS and MongoDB databases. To support that, we start with the core JPA and MongoDB frameworks using their associated starter artifacts.

RDBMS and MongoDB Core Dependencies

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb</artifactId>
</dependency>
```

286.2. Module Main Tree

The module's main tree contains a Spring MVC controller that will be injected with a RDBMS [DataSource](#) and [MongoClient](#). The other source files are there to supply a generic Spring Boot application.

Module Main Tree

```
src/main/java
`-- info
  '-- ejava
    '-- examples
      '-- svc
        '-- tcontainers
          '-- hello
            |-- TestcontainersHelloApp.java
            '-- controllers
              |-- ExceptionAdvice.java
              '-- HelloDBController.java ①
```

① controller injected with RDBMS [DataSource](#) and [MongoClient](#)

286.3. Injected RestController Component

We will use a familiar `@RestController` as an example server-side component to inject the lower-level `DataSource` and `MongoClient` components. The `DataSource` and `MongoClient` components encapsulate the database connections.

The database components are declared optional in the event that we are focusing on just RDBMS, MongoDB, or remote client in a specific test and do not need one or both database connections.

RestController Injected with DataSource and MongoClient Components

```
import com.mongodb.client.MongoClient;
import javax.sql.DataSource;
...
@RestController
@Slf4j
public class HelloDBController {
    private final DataSource dataSource;
    private final MongoClient mongoClient;
    ①
    public HelloDBController(@Autowired(required = false) DataSource dataSource,
                            @Autowired(required = false) MongoClient mongoClient) {
        this.dataSource = dataSource;
        this.mongoClient = mongoClient;
    }
}
```

- ① database components declared optional in event a specific test has not adequately configured the alternate



@RestController do not Perform Database Actions

`@RestController` components do not normally directly interact with a database. Their primary job is to interact with the web and accomplish tasks using injected service components. The example is thinned down to a single component—to do both/all—for simplicity.

286.3.1. RDBMS Connection Test Endpoint

The RDBMS path will make a connection to the database and return the resulting JDBC URL. A successful return of the expected URL will mean that the RDBMS connection was correctly established.

RDBMS Endpoint Tests Connection and Returns JDBC URL

```
@GetMapping(path="/api/hello/jdbc",
            produces = {MediaType.TEXT_PLAIN_VALUE})
public String helloDataSource() throws SQLException {
    try (Connection conn = dataSource.getConnection()) {
        return conn.getMetaData().getURL();
    }
}
```

```
}
```

286.3.2. MongoDB Connection Test Endpoint

The MongoDB path will return a cluster description that identifies the database connection injected. However, to actually test the connection—we are requesting a list of database names from MongoDB. The cluster description does not actively make a connection.

MongoDB Endpoint Tests Connection and Returns Cluster Description

```
@GetMapping(path="/api/hello/mongo",
    produces = {MediaType.TEXT_PLAIN_VALUE})
public String helloMongoClient() {
    log.info("dbNames: ", mongoClient.listDatabaseNames().first()); //test connection
    return mongoClient.getClusterDescription().getShortDescription();
}
```

286.4. In-Memory Example

I will start the demonstration with the RDBMS path and the H2 in-memory database to cover the basics.

286.4.1. Maven Dependencies

I have already shown the required JPA dependencies. For this test, we need to add the H2 database as a test dependency.

RDBMS and H2 Test Maven Dependencies

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>test</scope> ①
</dependency>
```

① H2 is used only in tests

286.4.2. Unit Integration Test Setup

The following snippet shows the unit integration test (NTest) setup using the in-memory H2 database in near full detail. The expected JDBC URL and actual DataSource is injected along with setup to communicate with the `@RestController`.

Unit Integration Test Core Setup

```
@SpringBootTest(classes={TestcontainersHelloApp.class,
    ClientNTestConfiguration.class},②
    properties={"spring.datasource.url=jdbc:h2:mem:testcontainers"},①
```

```

    webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
@ActiveProfiles({"test"})
@Slf4j
class HelloH2InMemoryNTTest {
    @Value("${spring.datasource.url}")
    private String expectedJdbcUrl; ③
    @Autowired
    private DataSource dataSource; ②
    @Autowired
    private RestTemplate anonymousUser;
    private UriBuilder dbUrl;

    @BeforeEach
    void init(@LocalServerPort int port) {
        dbUrl = UriComponentsBuilder.fromHttpUrl("http://localhost").port(port).path(
        "/api/hello/{db}");
    }
}

```

- ① provides a concrete database URL for this test
- ② configuration declares the injected RestTemplate
- ③ DataSource will represent connection to provided URL property

Known, non-Dynamic Property values can be Declared Using Simple Properties



Notice the `spring.data.source.url` value is constant and was known well in advance of running the JUnit JVM. This allows it to be defined as a static property. That won't always be the case and will address my dynamic cases shortly, within this lecture.

286.4.3. Inject DataSource into Test

The following test verifies that the JDBC URL provided by the `DataSource` will be the same as the `spring.datasource.url` property injected. This verifies our test can connect to the database and satisfy the authentication requirements (none in this in-memory case).

Verify Connection using DataSource

```

@Test
void can_get_connection() throws SQLException {
    //given
    then(dataSource.isNotNull());
    String jdbcUrl;
    //when
    try(Connection conn=dataSource.getConnection()) {
        jdbcUrl=conn.getMetaData().getURL();
    }
    //then
    then(jdbcUrl).isEqualTo(expectedJdbcUrl);
}

```

286.4.4. Inject DataSource into Server-side

The next test verifies that the server-side component under test (`@RestController`) was injected with a `DataSource`, can complete a connection, and returns the expected JDBC URL.

Verify Server-side Connection

```
@Test
void server_can_get_jdbc_connection() {
    //given
    URI url = dbUrl.build("jdbc");
    RequestEntity<Void> request = RequestEntity.get(url).build();
    //when
    ResponseEntity<String> response = anonymousUser.exchange(request, String.class);
    //then
    String jdbcUrl=response.getBody();
    then(jdbcUrl).isEqualTo(expectedJdbcUrl);
```

At this point we have a pretty straight forward unit integration framework that is similar to ones we have used in past lectures. Lets move onto using Postgres and leverage Testcontainers to more easily integrate the Postgres Docker image.

Chapter 287. Postgres Container Example

At this point we want to take our RDBMS development and testing to the next level and work with a real RDBMS instance of our choosing - Postgres.

287.1. Maven Dependencies

The following snippet lists the primary Testcontainers Maven dependencies. Artifact versions are defined in the `testcontainers-bom`, which is automatically imported by the `spring-boot-dependencies` BOM. We add the `testcontainers` dependency for core library calls and `GenericContainer` management. We add the `junit-jupiter` artifact for the JUnit-specific integration. They are both defined with `scope=test` since there is no dependency between our production code and the Testcontainers.

Testcontainers Maven Dependencies

```
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>testcontainers</artifactId> ①
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>junit-jupiter</artifactId> ②
    <scope>test</scope>
</dependency>
```

① core Testcontainers calls

② JUnit-specific calls

The Postgres database driver is added as a `scope=runtime` dependency so that we can communicate with the Postgres database instance during deployment and testing. The Postgres-specific Testcontainers artifact is added as a dependency to supply a convenient API for obtaining a running Postgres Docker image. Its `scope=test` restricts it only to testing.

Postgres/Testcontainers Dependencies

```
<dependency> ①
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <scope>runtime</scope>
</dependency>

<dependency> ②
    <groupId>org.testcontainers</groupId>
    <artifactId>postgresql</artifactId>
    <scope>test</scope>
</dependency>
```

① needed during deployment and tests

② needed during test only

287.2. Unit Integration Test Setup

The following shows the meat of how we setup our Postgres container and integrate it into our RDBMS unit integration test. There is a lot to conceptually unpack and will do so in the follow-on sections below.

PostgreSQLContainer Test Setup

```
@SpringBootTest ...
@Testcontainers //manages lifecycle of container
@Slf4j
class HelloPostgresTestcontainerNTest {
    @Container
    private static PostgreSQLContainer postgres = new PostgreSQLContainer<>(
        "postgres:12.3-alpine");
    @DynamicPropertySource
    private static void addLateSpringContextProperties(DynamicPropertyRegistry
registry) {
        registry.add("spring.datasource.url", () -> postgres.getJdbcUrl());
        registry.add("spring.datasource.username", () -> postgres.getUsername());
        registry.add("spring.datasource.password", () -> postgres.getPassword());
    }

    @Value("${spring.datasource.url}")
    private String expectedJdbcUrl;
    @Autowired
    private DataSource dataSource;
```

287.2.1. PostgreSQLContainer Setup

Testcontainers provides a `GenericContainer` that is what it sounds like. It generically manages the download, lifecycle, and communication of a Docker image. However, Testcontainers provides many `container-specific "modules"` that extend `GenericContainer` and encapsulate most of the module-specific setup work for you.

Since this lecture is trying to show you the "easy street" to integration, we'll go the module-specific `PostgreSQLContainer` route. All we need to do is

- specify a specific image name if we do not want to accept the default.
- declare the container using a static variable so that it is instantiated before the tests
- annotate the container with `@Container` to identify it to be managed
- annotate the test case with `@Testcontainers` to have the library automatically manage the lifecycle of the container(s).

Container Instantiated during Static Initialization

```
import org.testcontainers.junit.jupiter.Container;
import org.testcontainers.junit.jupiter.Testcontainers;
import org.testcontainers.containers.PostgreSQLContainer;
...
@Testcontainers //manages lifecycle of container ③
class HelloPostgresTestcontainerNTest {
    @Container ②
    private static PostgreSQLContainer postgres = new PostgreSQLContainer<>(
        "postgres:12.3-alpine"); ①
```

① declare container using a static variable

② `@Container` identifies container(s) for Testcontainers to manage

③ `@Testcontainers` annotation activates management that automatically starts/stops annotated containers

287.2.2. PostgresSQLContainer Startup

With just that in place (and the Docker daemon running), we can attempt to run the tests and see the container lifecycle in action.

- Testcontainers first starts a separate `Ryuk` container that will help manage the containers externally. The `Ryuk` container is primarily used to perform cleanup at the end of the tests.
- if required, the `postgres` Docker image is automatically downloaded
- the `postgres` Docker container is started
- a connection to the running `postgres` Docker container is made available to the host JVM (`jdbc:postgresql://localhost:55747/test`)

```
DockerClientProviderStrategy -- Loaded
org.testcontainers.dockerclient.UxixSocketClientProviderStrategy from
~/.testcontainers.properties, will try it first
DockerClientProviderStrategy -- Found Docker environment with local Unix socket
(unix:///var/run/docker.sock)

DockerClientFactory -- Docker host IP address is localhost
DockerClientFactory -- Connected to docker:
...
①
ryuk:0.5.1 -- Creating container for image: testcontainers/ryuk:0.5.1
ryuk:0.5.1 -- Container testcontainers/ryuk:0.5.1 is starting:
b8a3a26580339bb98445f160610db339834216dfc1403f5e1a2b99800feb1a43
ryuk:0.5.1 -- Container testcontainers/ryuk:0.5.1 started in PT5.450402S
utility.RyukResourceReaper -- Ryuk started - will monitor and terminate Testcontainers
containers on JVM exit
...
②
tc.postgres:12.3-alpine -- Pulling docker image: postgres:12.3-alpine. Please be
patient; this may take some time but only needs to be done once.
```

```

tc.postgres:12.3-alpine -- Starting to pull image
tc.postgres:12.3-alpine -- Pulling image layers: 0 pending, 0 downloaded, 0
extracted, (0 bytes/0 bytes)
tc.postgres:12.3-alpine -- Pulling image layers: 7 pending, 1 downloaded, 0
extracted, (1 KB/? MB)
...
tc.postgres:12.3-alpine -- Pull complete. 8 layers, pulled in 5s (downloaded 55 MB at
11 MB/s)

③
tc.postgres:12.3-alpine -- Creating container for image: postgres:12.3-alpine
tc.postgres:12.3-alpine -- Container postgres:12.3-alpine is starting:
78c1eda9cd432bb1fa434d65db1c4455977e0a0975de313eea9af30d09795caa
tc.postgres:12.3-alpine -- Container postgres:12.3-alpine started in PT1.913888S
tc.postgres:12.3-alpine -- Container is started (JDBC URL:
jdbc:postgresql://localhost:55747/test?loggerLevel=OFF) ④

Starting HelloPostgresTestcontainerNTest using Java 17.0.3 with PID 1853 (started by
jim in testcontainers-ntest-example) ⑤
...

```

- ① Testcontainers starts separate **Ryuk** container to manage container cleanup
- ② if required, the Docker image is automatically downloaded
- ③ the Docker container is started
- ④ connection(s) to the running Docker container made available to host JVM
- ⑤ test(s) start

If that is all we provided, we would see traces of the error shown in the following snippet—indicating we are missing configuration properties providing connection information needed to construct the **DataSource**.

Spring Requires DataSource Properties

```

Error creating bean with name
'info.ejava.examples.svc.tcontainers.hello.HelloPostgresTestcontainerNTest': Injection
of autowired dependencies failed
...
Could not resolve placeholder 'spring.datasource.url' in value
"${spring.datasource.url}" ①

```

- ① we do not know this value at development or JVM startup time

287.2.3. DynamicPropertyRegistry

Commonly we will have the connection information known at the start of heavyweight Maven Failsafe integration tests that is passed in as a concrete property. That would have been established during the **pre-integration-test** phase. In this case, the JVM and JUnit have already started and it is too late to have a property file in place with these dynamic properties.

A clean solution is to leverage Spring's `DynamicPropertyRegistry` test construct and define them at runtime.

- declare a static method that accepts a single `DynamicPropertyRegistry`
- annotate the static method with `@DynamicPropertySource`
- populate the `DynamicPropertyRegistry` with properties using a `Supplier<Object>` lambda function

Dynamically Define DataSource Properties from Container at Runtime

```
import org.springframework.test.context.DynamicPropertyRegistry;
import org.springframework.test.context.DynamicPropertySource;
...
class HelloPostgresTestcontainerNTest {
    @Container
    private static PostgreSQLContainer postgres = ...
    @DynamicPropertySource ①
    private static void addLateSpringContextProperties(DynamicPropertyRegistry
registry) {
        registry.add("spring.datasource.url", () -> postgres.getJdbcUrl()); ②
        registry.add("spring.datasource.username", () -> postgres.getUsername());
        registry.add("spring.datasource.password", () -> postgres.getPassword());
    }
}
```

① `@DynamicPropertySource` identifies static methods that work with the `DynamicPropertyRegistry`

② defines property supplier methods that can produce the property value at runtime

Spring added DynamicPropertySource to Support Testcontainers



The `DynamicPropertySource` Javadoc states this test construct was originally added in Spring 5.2.5 to explicitly support testing with Testcontainers (but can be used for anything).

With the `PostgreSQLContainer` instance started and `@DynamicPropertySource` in place, the properties are successfully injected into the test.

Dynamic Property Injection

```
@Value("${spring.datasource.url}")
private String expectedJdbcUrl;

@Test
void can_populate_spring_context_with_dynamic_properties() {
    then(expectedJdbcUrl).matches(
        "jdbc:postgresql://(:localhost|host.docker.internal):[0-9]+/test.*");
}
```

287.3. Inject Postgres Container DataSource into Test

Like with the in-memory H2 database test, we are able to confirm that the injected `DataSource` is able to establish a connection to the Postgres database running within the container managed by Testcontainers.

Verify Connection using DataSource

```
@Test  
void can_get_connection() throws SQLException {  
    //given  
    then(dataSource.isNotNull());  
    Connection conn= dataSource.getConnection();  
    //when  
    String jdbcUrl;  
    try (conn) {  
        jdbcUrl= conn.getMetaData().getURL();  
    }  
    //then  
    then(jdbcUrl).isEqualTo(expectedJdbcUrl); ①  
}
```

① `expectedJdbcUrl` already matches `jdbc:postgresql://(:localhost|host.docker.internal):[0-9]+/test.*`

287.4. Inject Postgres Container DataSource into Server-side

We have also successfully injected the `DataSource` into the server-side component (which is running in the same JVM as the JUnit test).

Verify Server-side Connection

```
@Test  
void server_can_get_jdbc_connection() {  
    //given  
    URI url = dbUrl.build("jdbc");  
    RequestEntity<Void> request = RequestEntity.get(url).build();  
    //when  
    ResponseEntity<String> response = anonymousUser.exchange(request, String.class);  
    //then  
    then(response.getStatusCode()).isEqualTo(HttpStatus.OK);  
    String jdbcUrl= response.getBody();  
    then(jdbcUrl).isEqualTo(expectedJdbcUrl);  
}
```

287.5. Runtime Docker Containers

The following snippet of a `docker ps` command during the test shows the Testcontainers `Ryuk` container running along side our `postgres` container. The ports for both are dynamically assigned.

docker ps Output

IMAGE	PORTS	NAMES
<code>postgres:12.3-alpine</code>	<code>0.0.0.0:55747->5432/tcp</code>	<code>nervous_benz</code> ①
<code>testcontainers/ryuk:0.5.1</code>	<code>0.0.0.0:55742->8080/tcp</code>	<code>testcontainers-ryuk-6d7a4199-9471-4269-...</code>

① postgres 5432 SQL port is exposed to localhost using a dynamically assigned port number

Recall that Testcontainers supplied the `postgres` container port to our test using the module-specific `getJdbcUrl()` method.

Module-specific Containers Provide Convenience Methods

```
@Container
private static PostgreSQLContainer postgres = ...
@DynamicPropertySource
private static void addLateSpringContextProperties(DynamicPropertyRegistry registry) {
    registry.add("spring.datasource.url", () -> postgres.getJdbcUrl()); ①
...
}
```

① contained dynamically assigned port number (55747)

Chapter 288. MongoDB Container Example

We next move on to establish an integration test using MongoDB and Testcontainers. Much of the [MongoDBContainer](#) concepts are the same as with [PostgresSQLContainer](#), except the expected property differences and a configuration gotcha.

288.1. Maven Dependencies

The following snippet shows the additional MongoDB module dependency we need to add on top of the [spring-boot-starter-data-mongodb](#), [testcontainers](#), and [junit-jupiter](#) dependencies we added earlier.

Additional Testcontainers MongoDB Module Dependency

```
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>mongodb</artifactId>
    <scope>test</scope>
</dependency>
```

288.2. Unit Integration Test Setup

The following snippet shows the meat of the overall [MongoDBContainer](#) setup. As with [PostgresSQLContainer](#), we have the choice to use the [GenericContainer](#) or the module-specific [MongoDBContainer](#). However, with this database I ran into a time-consuming gotcha that might influence choices.

MongoDB started supporting multi-document transactions with version 4.0. However, to use transactions—one must define a "replica set" (just consider that a "setting"). Wanting to be full featured for testing, [MongoDBContainer](#) configures MongoDB for a (single-node) replica set. However, that is performed using an admin-level command issued to the just started MongoDB database instance. That command fails if authentication is enabled because the replica set admin command from the [MongoDBContainer](#) is issued without credentials. If that command fails, [MongoDBContainer](#) considers the Docker container startup as failed and terminates.

I will demonstrate how to run the [MongoDBContainer](#), enabling transactions by disabling authentication. I will also demonstrate how to run MongoDB image within the [GenericContainer](#), enabling security and no support for transactions. Both, solely have the goal to complete a connection as part of this lecture. I am not demonstrating any database-specific capabilities beyond that.

MongoDBContainer Test Setup

```
@SpringBootTest ...
@Testcontainers //manages lifecycle of container
@Slf4j
class HelloMongoDBTestcontainerNTest {
    @Container
```

```

private static MongoDBContainer mongoDB = new MongoDBContainer("mongo:4.4.0-
bionic");
@DynamicPropertySource
private static void addLateSpringContextProperties(DynamicPropertyRegistry
registry) {
    registry.add("spring.data.mongodb.uri", () -> mongoDB.getReplicaSetUrl(
"testcontainers"));
}

@Value("${spring.data.mongodb.uri}")
private String expectedMongoUrl;
@Autowired
private MongoClient mongoClient;

```

① do not enable authentication with `MongoDBContainer`

288.2.1. MongoDBContainer Setup

Like with the `PostgresSQLContainer`, we

- specify the name of the MongoDB image
- instantiate the `MongoDBContainer` during static initialization
- annotate the container with `@Container` to identify it to be managed
- annotate the test case with `@Testcontainers` to have the library automatically manage the lifecycle of the container(s).

Also of important note — `MongoDBContainer` does not enable authentication and we **must not** enable it by setting environment variables

- `MONGO_INITDB_ROOT_USERNAME`
- `MONGO_INITDB_ROOT_PASSWORD`

MongoDBContainer does not support Authentication

```

import org.testcontainers.containers.MongoDBContainer;
import org.testcontainers.junit.jupiter.Container;
import org.testcontainers.junit.jupiter.Testcontainers;

@Testcontainers //manages lifecycle of container
class HelloMongoDBTestcontainerNTest {
    @Container
    private static MongoDBContainer mongoDB = new MongoDBContainer("mongo:4.4.0-
bionic")
    // .withEnv("MONGO_INITDB_ROOT_USERNAME", "admin") ①
    // .withEnv("MONGO_INITDB_ROOT_PASSWORD", "secret")
    ;
}

```

① setting username/password enables MongoDB authentication and causes `MongoDBContainer` ReplicaSet configuration commands to fail

288.2.2. GenericContainer Setup

We can alternatively use the `GenericContainer` and push all the levers we want. In the following example, we will instantiate a MongoDB Docker container that supports authentication but has not been setup with a replica set to support transactions.

Can enable MongoDB Authentication with GenericContainer

```
import org.testcontainers.containers.GenericContainer;
import org.testcontainers.junit.jupiter.Container;
import org.testcontainers.junit.jupiter.Testcontainers;

@Testcontainers //manages lifecycle of container
class HelloMongoDBGenericContainerNTest {
    @Container
    private static GenericContainer mongoDB = new GenericContainer("mongo:4.4.0-
bionic")
        .withEnv("MONGO_INITDB_ROOT_USERNAME", "admin")
        .withEnv("MONGO_INITDB_ROOT_PASSWORD", "secret")
        .withExposedPorts(27017);
```

288.2.3. MongoDB Dynamic Properties

At this point, we need to locate and express the MongoDB URI. The technique will be slightly different for `MongoDBContainer` and `GenericContainer` because only the `MongoDBContainer` knows what a MongoDB URI is.

The snippet below shows how we can directly define the MongoDB URI property using the `MongoDBContainer.getReplicaSetUrl()`. The resulting MongoDB URI will be `mongodb://localhost:63216/testcontainers;`

Obtain MongoDB URI from MongoDBContainer

```
@DynamicPropertySource
private static void addLateSpringContextProperties(DynamicPropertyRegistry registry) {
    registry.add("spring.data.mongodb.uri", () -> mongoDB.getReplicaSetUrl(
        "testcontainers"));
}
```

The snippets below show how we can assemble the MongoDB URI using properties that the `GenericContainer` knows about (e.g., hostname, port) and what we know we provided to the `GenericContainer`.

Obtain MongoDB URI from GenericContainer

```
@DynamicPropertySource
private static void addLateSpringContextProperties(DynamicPropertyRegistry registry) {
    String userName = (String) mongoDB.getEnvMap().get("MONGO_INITDB_ROOT_USERNAME");
①
```

```

        String password = (String) mongoDB.getEnvMap().get("MONGO_INITDB_ROOT_PASSWORD");
        registry.add("spring.data.mongodb.uri", () ->
            ClientNTTestConfiguration.mongoUrl(userName, password,
                mongoDB.getHost(), mongoDB.getMappedPort(27017), "testcontainers"));
    }
}

```

- ① `userName` and `password` were supplied as environment variables when `GenericContainer` created

With `userName`, `password`, host, port, and database name—we can manually create a MongoDB URI.

Format MongoDB URI from Properties

```

public static String mongoUrl(String userName, String password, String host, int port,
String database) {
    return String.format("mongodb://$s:$s@$s:$d/$s?authSource=admin", userName,
    password, host, port, database);
}

```

The resulting MongoDB URI will be
`mongodb://admin:secret@localhost:54568/testcontainers?authSource=admin;`

288.3. Inject MongoDB Container Client into Test

With our first test, we are able to verify that we can inject a `MongoClient` into the test case and that it matches the expected MongoDB URI. We also need to execute additional command(s) to verify the connection because just describing the cluster/connection does not establish any communication between the `MongoClient` and MongoDB.

Verify Connection using MongoClient

```

@Test
void can_get_connection() {
    //given
    then(mongoClient).isNotNull();
    //when
    String shortDescription = mongoClient.getClusterDescription().getShortDescription();
    //then
    new MongoVerifyTest().actual_hostport_matches_expected(expectedMongoUrl,
    shortDescription); ①
    then(mongoClient.listDatabaseNames()).contains("admin"); ②
}

```

- ① host:port will get extracted from the Mongo URI and cluster description and compared

- ② `databaseNames()` called to test connection

The following snippet shows an extraction and comparison of the host:port values from the Mongo

URI and cluster description values.

Verify Connection Matches Injected Value

```
//..., servers=[{address=localhost:56295, type=STANDALONE...  
private static final Pattern DESCRIPTOR_ADDRESS_PATTERN = Pattern.compile("address=(["A-Za-  
z\\.\\d]+)");  
//mongodb://admin:secret@localhost:27017/testcontainers  
private static final Pattern URL_HOSTPORT_PATTERN = Pattern.compile("[@/](["A-Za-z  
\\.\\d]+)/*");  
  
void actual_hostport_matches_expected(String expectedMongoUrl, String description) {  
    Matcher m1 = DESCRIPTOR_ADDRESS_PATTERN.matcher(description);  
    then(expectedMongoUrl).matches(url->m1.find(), DESCRIPTOR_ADDRESS_PATTERN.toString())  
;①  
  
    Matcher m2 = URL_HOSTPORT_PATTERN.matcher(expectedMongoUrl);  
    then(expectedMongoUrl).matches(url->m2.find(), URL_HOSTPORT_PATTERN.toString());①  
  
    then(m1.group(1)).isEqualTo(m2.group(1)); ②  
}  
}
```

① extracting host:port values from sources

② comparing extracted host:port values

288.4. Inject MongoDB Container Client into Server-side

This next snippet shows we can verify the server-side was injected with a `MongoClient` that can communicate with the expected database.

Verify Server-side Connection

```
@Test  
void server_can_get_mongo_connection() {  
    //given  
    URI url = dbUrl.build("mongo");  
    RequestEntity<Void> request = RequestEntity.get(url).build();  
    //when  
    String shortDescription = anonymousUser.exchange(request, String.class).getBody();  
    //then  
    new MongoVerifyTest().actual_hostport_matches_expected(expectedMongoUrl,  
    shortDescription);  
}
```

288.5. Runtime Docker Containers

The following snippet of a docker ps command during the test shows the Testcontainers Ryuk

container running along side our `mongo` container. The ports for both are dynamically assigned.

docker ps Output

IMAGE	CREATED	PORTS	NAMES
<code>mongo:4.4.0-bionic</code>	1 second ago	<code>0.0.0.0:59627->27017/tcp</code>	<code>gifted_hoover</code>
<code>testcontainers/ryuk:0.5.1</code>	1 second ago	<code>0.0.0.0:59620->8080/tcp</code>	<code>testcontainers-ryuk-a6c35bc3-66d7-...</code>

Chapter 289. Docker Compose Example

The previous two examples represent the heart of what Testcontainers helps us achieve—easy integration with back-end resources for testing. Testcontainers can help automate many more tasks, including interfacing with Docker Compose. In the following example, I will use a familiar Docker Compose setup to demonstrate:

- Testcontainers interface with Docker Compose
- obtain container connections started using Docker Compose
- build and test the server-side application under test using Testcontainers and Docker Compose

289.1. Maven Aspects

The `DockerComposeContainer` is housed within the primary `org.testcontainers:testcontainers` dependency. There are no additional Maven dependencies required.

However, our JUnit test(s) will need to run after the Spring Boot Executable JAR has been created, so we need to declare the test with an `IT` suffix and add the Maven Failsafe plugin to run the test during the `integration-test` phase. Pretty simple.

Run Test(s) During integration-test Phase using Maven Failsafe Plugin

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-failsafe-plugin</artifactId>
      <executions>
        <execution>
          <id>integration-test</id>
          <goals>
            <goal>integration-test</goal>
          </goals>
        </execution>
        <execution>
          <id>verify</id>
          <goals>
            <goal>verify</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
  ...

```

289.2. Example Files

The following snippet shows our basic module structure where we have

- Docker/Docker Compose resource files intended for production
- Docker/Docker Compose resource files used for testing
- an integration test

The dual Docker Compose files is not required. This is only an example of using multiple files and if you need something unique for testing. Our JUnit test is defined as a Maven Failsafe IT test so that the Spring Boot Executable JAR can be built in time to run the test. We will not need the pre/post-integration-test phases.

```
src/main
|-- docker ①
|   |-- Dockerfile
|   |-- docker-compose.yml
|   `-- run_env.sh
...
src/test
|-- java
|   '-- info
|       '-- ejava
|           '-- examples
|               '-- svc
|                   '-- tcontainers
|                       '-- hello
|                           |-- ClientNTestConfiguration.java
|                           |-- HelloApiContainerIT.java
|
`-- resources
    |-- docker-compose-test.yml ②
    ...
...
```

① production Docker/Docker Compose configuration

② Docker Compose configuration settings for test

If we test within the IDE, the Spring Boot Executable JAR must be available to build the API's Docker image. The Spring Boot Executable JAR can be re-built using the following commands

Rebuild the Spring Boot Executable JAR for API image

```
$ mvn clean package -DskipTests && ls target/*bootexec*
...
target/testcontainers-ntest-example-6.1.0-SNAPSHOT-bootexec.jar
```

289.2.1. Production Docker Compose File

The following snippet shows an example Docker Compose file playing the role of what we might run in production (minus the plaintext credentials). Notice that the

- Postgres and MongoDB ports are not externally defined. The Postgres and MongoDB ports are already defined by their Dockerfile definition. The API will be able to communicate with the databases using the internal Docker network created by Docker Compose.
- only the image identification is supplied for the API. The build commands will not be necessary or desired in production.

src/main/docker/docker-compose.yml

```
services:  
  postgres:  
    image: postgres:12.3-alpine  
    environment:  
      POSTGRES_PASSWORD: secret  
  mongodb:  
    image: mongo:4.4.0-bionic  
    environment:  
      MONGO_INITDB_ROOT_USERNAME: admin  
      MONGO_INITDB_ROOT_PASSWORD: secret  
  api: ①  
    image: testcontainers-ntest-example:latest  
    ports:  
      - "${HOST_API_PORT:-8080}:8080" ②  
    depends_on:  
      - postgres  
      - mongodb  
    environment:  
      - DATABASE_URL=postgres://postgres:secret@postgres:5432/postgres  
      -  
      MONGODB_URI=mongodb://admin:secret@mongodb:27017/testcontainers?authSource=admin
```

① only runtime commands supplied, build commands not desired here

② API defaults to exposed pre-defined port here

289.2.2. Test Docker Compose File

The snippet below shows a second Docker Compose file playing the role of the test overrides. Here we are defining the build commands the API image. Note that we need to set the Docker build for context to a high enough level to include the target tree with the Spring Boot Executable JAR.

src/test/resources/docker-compose-test.yml

```
services:  
  api:  
    build: #set build context to module root so that target/*bootexec.jar is within  
    view
```

```

context: ../../..
dockerfile: src/main/docker/Dockerfile
ports:
- "8080" #API port exposed on random host port

```

289.2.3. Dockerfile

The following snippet shows a familiar Dockerfile that will establish a layered Docker image. I have assigned a few extra variables (e.g., `BUILD_ROOT`, `RESOURCE_DIR`) that might be helpful building the image under different circumstances but were not needed here.

I have also included the use of a `run.sh` bash script in the event we need to add some customization to the API JVM. It is used in this example to convert provided environment variables into `DataSource` and `MongoClient` properties required by the server-side API.

src/main/docker/Dockerfile

```

FROM eclipse-temurin:17-jre AS builder
WORKDIR /builder
ARG BUILD_ROOT=.
ARG JAR_FILE=${BUILD_ROOT}/target/*-bootexec.jar
ARG RESOURCE_DIR=${BUILD_ROOT}/src/main/docker
COPY ${JAR_FILE} application.jar
RUN java -Djarmode=tools -jar application.jar extract --layers --launcher
--destination extracted
COPY ${RESOURCE_DIR}/run_env.sh .

FROM eclipse-temurin:17-jre
WORKDIR /application
COPY --from=builder /builder/extracted/dependencies/ ./
COPY --from=builder /builder/extracted/spring-boot-loader/ ./
COPY --from=builder /builder/extracted/snapshot-dependencies/ ./
COPY --from=builder /builder/extracted/application/ ./
COPY --chmod=555 --from=builder /builder/run_env.sh ./
#https://github.com/spring-projects/spring-boot/issues/37667
ENTRYPOINT ["../run_env.sh",
"java","org.springframework.boot.loader.launch.JarLauncher"]

```

289.3. Integration Test Setup

The snippet below shows the JUnit setup for the `DockerComposeContainer`.

- the test is named as a Maven Failsafe IT test. It requires no special input properties
- there is no web/server-side environment hosted within this JVM. The server-side will run within the API Docker image.
- details of the `DockerComposeContainer` and `DynamicPropertyRegistry` configuration are contained within the `ClientNTestConfiguration` class

- the test case is annotated with `@Testcontainers` to trigger the lifecycle management of containers
- the `DockerComposeContainer` is declared during static initialization and annotated with `@Container` so that its lifecycle gets automatically managed
- `@BeforeEach` initialization is injected with the host and port of the remote API running in Docker, which will not be known until the `DockerComposeContainer` started and the API Docker container is launched

```

import org.testcontainers.containers.DockerComposeContainer;
import org.testcontainers.junit.jupiter.Container;
import org.testcontainers.junit.jupiter.Testcontainers;

@SpringBootTest(classes= ClientNTTestConfiguration.class,
               webEnvironment = SpringBootTest.WebEnvironment.NONE)
@EnableAutoConfiguration
@Testcontainers //manages lifecycle for @Containers ①
class HelloApiContainerIT {
    @Container ①
    private static DockerComposeContainer network = ClientNTTestConfiguration
        .testEnvironment();
    @DynamicPropertySource ②
    private static void addLateSpringContextProperties(DynamicPropertyRegistry
registry) {
        ClientNTTestConfiguration.initProperties(registry, network);
    }

    @BeforeEach
    void init( ③
        @Value("${it.server.host:localhost}") String remoteApiContainerHost,
        @Value("${it.server.port:9090}") int remoteApiContainerPort) {
        dbUrl=UriComponentsBuilder.fromHttpUrl("http://localhost")
            .host(remoteApiContainerHost)
            .port(remoteApiContainerPort)
            .path("/api/hello/{db}");
    }
}

```

① `@Testcontainers` annotation will start/stop all Testcontainers marked with `@Container`

② `DynamicPropertyRegistry` will be used to supply required properties not known until Docker containers start

③ host and port of remote API will not be known until Docker containers start, thus come from `DynamicPropertyRegistry`

289.3.1. DockerComposeContainer Setup

With the added complexity of the `DockerComposeContainer` setup, I chose to move that to a sharable static helper method. There is again a lot to unpack here.

DockerComposeContainer Construction

```
public static DockerComposeContainer testEnvironment() {
    return new DockerComposeContainer("testcontainers-ntest", ①
        List.of(new File("src/main/docker/docker-compose.yml"), //core wiring ②
               new File("target/test-classes/docker-compose-test.yml")))) //build & port
info
    .withBuild(true) ③
    .withExposedService("api", 8080) ④
    .withExposedService("postgres", 5432)
    .withExposedService("mongodb", 27017)
    .withLocalCompose(true) ⑤
    //https://github.com/testcontainers/testcontainers-java/pull/5608
    .withOptions("--compatibility") //change dashes to underscores ⑥
                                    //when using local=true
                                    //so testcontainers can find the
container_name
    .withStartupTimeout(Duration.ofSeconds(100));
}
```

- ① identifier
- ② configuration files
- ③ force image build
- ④ Identify Exposed Ports
- ⑤ Local or container-based Docker Compose
- ⑥ V1 or V2 Docker Compose container naming syntax

Identifier

Like with the command line, we can supply an identifier (e.g., `testcontainers-ntest`) that will prefix each container name created (e.g., `testcontainers-ntestkxzcp_mongodb_1`). This is not unique by itself, but it does help describe where the container came from.

Example Supplying Identifier via Command Line

```
$ docker-compose ... -p testcontainers-ntest up
#docker ps --format '{{.Names}}'
testcontainers-ntest-api-1
testcontainers-ntest-postgres-1
testcontainers-ntest-mongodb-1
```

Configuration Files

We can express the configuration in one or more files. This example uses two.

Example Supplying Multiple Files via Command Line

```
$ docker-compose -f src/main/docker/docker-compose.yml -f src/test/resources/docker-
```

```
compose-test.yml ...
```

Force Image Build

The image build is being forced each time the test is run.

Example Forcing Build via Command Line

```
$ docker-compose ... -p testcontainers-ntest up --build
```

This, of course, depends on the Spring Boot Executable JAR to be in place in the **target** directory. This is why we named it as a Failsafe IT test, so that it will run after the Maven **package** phase.

Quick Build/Re-build API and Spring Boot Executable JAR

If you are running the IT test in the IDE and need to package or update the Spring Boot Executable JAR, you can use the following command.



```
$ mvn clean package -DskipTests
```

Identify Exposed Ports

We identify which ports in each service to expose to this JVM. Testcontainers will start an "ambassador" container ([alpine/socat](#)) to proxy commands between the JUnit JVM and the running containers. There will not be a direct connection between the JUnit JVM and Docker containers launched. There is no requirement to expose the service ports the host network. The "ambassador" container will proxy all communications between host and Docker network.

Docker Containers with "Ambassador"/Proxy and "Cleanup"

IMAGE	POR
testcontainers/ryuk:0.5.1	0.0.0.0:54186->8080/tcp
alpine/socat:1.7.4.3-r0	0.0.0.0:54193->2000/tcp, 0.0.0.0:54194->2001/tcp, 0.0.0.0:54195->2002/tcp ①
mongo:4.4.0-bionic	27017/tcp ②
postgres:12.3-alpine	5432/tcp ②
testcontainers-ntest-example:latest	0.0.0.0:8080->8080/tcp ③

① [alpine/socat](#) "ambassador" container establishes proxy connections to Docker Compose-launched services

② services are not exposed to host network by default—they are proxied by "ambassador"

③ API service was exposed to host network because of contents in configuration

Local or Container-based Docker Compose

The localCompose flag determines whether to use the binaries from the local host (true) or from a Testcontainers-supplied container implementing Compose V2 (false). I have set this to

`localCompose=true` because I found the launched Docker Compose container needs to successfully mount the source filesystem to read resources like the docker-compose.yml files. This does require Docker Compose binaries to be present in all development and CI/CD environments, but it is a much more flexible option to work with.

V1 or V2 Docker Compose Container Naming

I am launching Docker Compose with the [--compatibility mode](#), which impacts Testcontainers' ability to locate Docker images once launched within a CI/CD environment. Docker Compose V1 used underscores (`_`) to separate word tokens in the image name (e.g., `testcontainers-ntestkxzcpt_mongodb_1`). Underscore (`_`) is not a valid character in DNS names ^[1] so Docker Compose changed the delimiter to dashes (`-`) in V2 (e.g., `testcontainers-ntestgxxu16-mongodb-1`) so that the Docker name could be used as a DNS name.

Docker Compose V2 uses DNS-legal Dashes (-) for Word Token Separators

```
$ docker-compose -f src/main/docker/docker-compose.yml -f src/test/resources/docker-compose-test.yml up

#docker ps --format {{.Names}}
docker-api-1 ①
docker-mongodb-1
docker-postgres-1
```

① V2 mode uses DNS-legal dashes in names to make name DNS-usable

The problem arises when the Testcontainers [uses the V1 formatting strategy](#) to locate a name formatted using V2.

HelloApiContainerIT Failure Because Present Name Not Found

```
ERROR tc.alpine/socat:1.7.4.3-r0 -- Could not start container
org.testcontainers.containers.ContainerLaunchException: Aborting attempt to link to
container testcontainers-ntesthz5z8g_mongodb_1 as it is not running ①
```

① looking for V1 `testcontainers-ntesthz5z8g_mongodb_1` vs V2 `testcontainers-ntesthz5z8g-mongodb-1`

We can compensate for that using the Docker Compose V2 [--compatibility](#) flag.

Docker Compose V2 --compatibility Returns Word Token Separators to V1 Underscore(_)

```
$ docker-compose -f src/main/docker/docker-compose.yml -f src/test/resources/docker-compose-test.yml --compatibility up

#docker ps --format {{.Names}}
docker_api_1 ①
docker_mongodb_1
docker_postgres_1
```

① --compatibility mode uses V1 underscore (_) characters



Manual Commands were just Demonstrations for Clarity

The manual docker-compose commands shown above were to demonstrate how Testcontainers will translate your options to a Docker Compose command. You will not need to manually execute a Docker Compose command with Testcontainers.

289.3.2. Dynamic Properties

Once the Docker Compose services are running, we need to obtain URI information to the services. If you recall the JUnit test setup, we initially need the `it.server.host` and `it.server.port` properties to inject into `@BeforeEach` initialization method.

JUnit Test Setup Requires API host:port Properties

```
@BeforeEach
void init(@Value("${it.server.host:localhost}") String remoteApiContainerHost,
           @Value("${it.server.port:9090}") int remoteApiContainerPort) {
    dbUrl=UriComponentsBuilder.fromHttpUrl("http://localhost")
        .host(remoteApiContainerHost)
        .port(remoteApiContainerPort)
        .path("/api/hello/{db}");
}
```

As with the previous examples, we can obtain the runtime container properties and form/supply the application properties using a static method annotated with `@DynamicPropertySource` and taking a `DynamicPropertyRegistry` argument.`

Property Initialization was Delegated to Helper Method

```
@DynamicPropertySource
private static void addLateSpringContextProperties(DynamicPropertyRegistry registry) {
    ClientNTTestConfiguration.initProperties(registry, network); ①
}
```

① calls static helper method within configuration class

The example does the work within a static helper method in the configuration class. The container properties are obtained from the running `DockerComposeContainer` using accessor methods that accept the service name and targeted port number as identifiers. Notice that the returned properties point to the "ambassador"/socat proxy host/ports and not directly to the targeted containers.



getServiceHost() Ignores Port# Under the Hood

I noticed that port number was ignored by the underlying implementation for `getServiceHost()`, so I am simply passing a null for port.

Obtain Dynamic Properties from DockerComposeContainer

```
public static void initProperties(DynamicPropertyRegistry registry,
DockerComposeContainer network) {
    //needed for @Tests to locate API Server
    registry.add("it.server.port", ()->network.getServicePort("api", 8080)); //60010
②
    registry.add("it.server.host", ()->network.getServiceHost("api", null));
//localhost
...
//docker ps --format '{{.Names}}\t{{.Ports}}'
//testcontainers-socat...          0.0.0.0:60010->2000/tcp, ...
//testcontainers-ntesttn8uks_api_1  0.0.0.0:59998->8080/tcp ①
```

① socat port **60010** proxies API port **8080**

② socat proxy port **60010** supplied for `getServicePort()`

In the event we want connections to the back-end databases, we will need to provide the standard `spring.datasource.*` and `spring.data.mongodb.uri` properties.

Injecting Back-end Database Services

```
@Autowired //optional -- just demonstrating we have access to DB
private DataSource dataSource;
@Autowired //optional -- just demonstrating we have access to DB
private MongoClient mongoClient;
```

We can obtain those properties from the running `DockerComposeContainer` and supply values we know from the Docker Compose files.

Obtain Dynamic Properties for Back-end Databases

```
//optional -- only if @Tests directly access the DB
registry.add("spring.data.mongodb.uri",()> mongoUrl("admin", "secret",
    network.getServiceHost("mongodb", null),
    network.getServicePort("mongodb", 27017), //60009 ①
    "testcontainers"
));
registry.add("spring.datasource.url",()>jdbcUrl(
    network.getServiceHost("postgres", null),
    network.getServicePort("postgres", 5432) //60011 ②
));
registry.add("spring.datasource.driver-class-name",()>"org.postgresql.Driver");
registry.add("spring.datasource.username",()>"postgres");
registry.add("spring.datasource.password",()>"secret");

//docker ps --format '{{.Names}}\t{{.Ports}}'
//testcontainers-socat...          ..., 0.0.0.0:60011->2001/tcp, 0.0.0.0:60009-
>2002/tcp
```

```
//testcontainers-ntesttn8uks_mongodb_1    27017/tcp ①  
//testcontainers-ntesttn8uks_postgres_1      5432/tcp ②
```

① socat port **60009** proxies mongodb port **27017**

② socat port **60011** proxies postgres port **5432**

One last set of helper methods assemble the runtime/dynamic container properties and form URLs to be injected as mandatory database component properties.

Database URI Helper Methods

```
public static String mongoUrl(String userName, String password, String host, int port,  
String database) {  
    return String.format("mongodb://$s:$s@$s:$d/$s?authSource=admin", userName,  
password, host, port, database);  
}  
public static String jdbcUrl(String host, int port) {  
    return String.format("jdbc:postgresql://$s:$d/postgres", host, port);  
}
```

With the test setup complete, we are ready to verify our

- test can communicate with the databases
- test can communicate with the server-side API
- the server-side API can communicate with the databases

289.4. Inject Postgres Connection into Test

The test below shows that the JUnit test can obtain a connection to the database via the locally-injected **DataSource** and verifies it is a Postgres instance. The port will not be **5432** in this case because the JUnit test is remote from the Docker Compose containers and communicates to each of them using the **socat** proxy.

Inject Postgres Connection into Test

```
@Test  
void dataSource_can_provide_connection() throws SQLException {  
    //given  
    then(dataSource.isNotNull());  
    Connection conn=dataSource.getConnection();  
    //when  
    String jdbcUrl;  
    try (conn) {  
        jdbcUrl=conn.getMetaData().getURL();  
    }  
    //then  
    then(jdbcUrl).contains("jdbc:postgresql")  
        .doesNotContain("5432"); //test uses socat proxy;
```

```

}
//testcontainers-ntestvgbpgg_postgres_1      5432/tcp
//testcontainers-socat-dzm9Rn9w                0.0.0.0:65149->2001/tcp, ...
//jdbc:postgresql://localhost:65149/postgres ①

```

① test obtains `DataSource` connection via `socat` proxy port

289.5. Inject Postgres Connection into Server-Side

The next test contacts the server-side API to obtain the JDBC URL of its `DataSource`. The server-side API will use port `5432` because it is a member of the Docker network setup by Docker Compose and configured to use `postgres:5432` via the `docker-compose.yml` environment variable (`DATABASE_URL=postgres://postgres:secret@postgres:5432/postgres`).

Inject Postgres Connection into Server-Side

```

@Test
void server_can_get_jdbc_connection() {
    //given
    URI url = dbUrl.build("jdbc");
    RequestEntity<Void> request = RequestEntity.get(url).build();
    //when
    String jdbcUrl = anonymousUser.exchange(request, String.class).getBody();
    //then
    //hostname will be postgres and port will be default internal 5432
    then(jdbcUrl).contains("jdbc:postgresql","postgres:5432");
}
//testcontainers-ntestvgbpgg_api_1      0.0.0.0:65145->8080/tcp
//testcontainers-ntestvgbpgg_postgres_1  5432/tcp
//jdbc:postgresql://postgres:5432/postgres ①

```

① server-side API uses local Docker network for `DataSource` connection

The specific URL was provided as an environment variable in the Docker Compose configuration.

docker-compose.yml Server-side Environment Variables

```

services:
  api:
    environment:
      - DATABASE_URL=postgres://postgres:secret@postgres:5432/postgres
      - ...

```

289.6. Inject MongoClient into Test

The snippet below shows that the JUnit test can obtain a connection to MongoDB via the locally-injected `MongoClient`. The port will not be `27017` in this case either because the JUnit test is remote from the Docker Compose containers and communicates to each of them using the `socat` proxy.

Inject MongoClient into Test

```
@Test
void mongoClient_can_get_connection() {
    //given
    then(mongoClient).isNotNull();
    //then
    then(mongoClient.getClusterDescription().getShortDescription())
        .doesNotContain("27017");
    then(mongoClient.listDatabaseNames()).contains("admin");
}
//{type=STANDALONE, servers=[{address=localhost:65148, ...
//testcontainers-socat-dzm9Rn9w          0.0.0.0:65148->2000/tcp
//testcontainers-ntestvgbpgg_mongodb_1    27017/tcp
```

289.7. Inject MongoClient into Server-side

The snippet below shows that the server-side API can obtain a connection to MongoDB via its injected `MongoClient` and will use hostname `mongodb` and port `27017` because it is on the same local Docker network as MongoDB and was configured to do so using the `docker-compose.yml` environment variable (

`MONGODB_URI=mongodb://admin:secret@mongodb:27017/testcontainers?authSource=admin`).

Inject MongoClient into Server-side

```
@Test
void server_can_get_mongo_connection() {
    //given
    URI url = dbUrl.build("mongo");
    RequestEntity<Void> request = RequestEntity.get(url).build();
    //when
    String shortDescription = anonymousUser.exchange(request, String.class).getBody();
    //then
    //hostname will be mongo and port will be default internal 27017
    then(shortDescription).contains("address=mongodb:27017");
}
//{type=STANDALONE, servers=[{address=mongodb:27017, ...
//testcontainers-ntestvgbpgg_api_1          0.0.0.0:65145->8080/tcp
//testcontainers-ntestvgbpgg_mongodb_1      27017/tcp
```

docker-compose.yml Server-side Environment Variables

```
services:
  api:
    environment:
      - ...
      -
MONGODB_URI=mongodb://admin:secret@mongodb:27017/testcontainers?authSource=admin
```


Chapter 290. Summary

In this module, we learned:

- how to more seamlessly integrate back-end resources into our tests using Docker, Docker Compose, and Testcontainers using Testcontainers library
- how to inject dynamically assigned properties into the application context of a test to allow them to be injected into components at startup
- to establish client connection to back-end resources from our JUnit JVM operating the unit test

Although integration tests should never fully replace unit tests, the capability demonstrated in this lecture shows how we can create very capable end-to-end tests to verify the parts will come together correctly. More features exist within Testcontainers than were covered here. Some examples include

- waitingFor strategies that help determine when the container is ready for use
- exec commands into container that could allow us to issue CLI database commands if helpful. Remember the `MongoDBContainer` issues the replicaSet commands using the Mongo CLI interface. It is an example of a container exec command.

```
log.debug("Initializing a single node node replica set...");  
Container.ExecResult execResultInitRs = this.execInContainer(this  
.buildMongoEvalCommand("rs.initiate();"));  
...  
private String[] buildMongoEvalCommand(String command) {  
    return new String[]{"sh", "-c", "mongosh mongo --eval \"\" + command + "\\" + ||  
    mongo --eval \"\" + command + "\""};  
}
```

AutoRentals Assignment 4: Integration Testing

copyright © 2024 jim stafford (jim.stafford@jhu.edu)

This assignment is targeted at developing an integration test that involves the management of a separate server process. The aspects under test should already be implemented in your assignment 3 solution. The focus of this assignment will be on testing a different way—one that is closer to the deployment option versus some of the more performant unit and unit integration options achieved earlier.

You have the option to implement the integration test in one of three ways

- a. Testing against a Spring Boot Executable JAR.
 - This involves a fair amount of Maven plugin configuration.
- b. Testing against a Docker Container using Docker/Maven plugins
 - This also involves a fair amount of Maven plugin configuration, adds extra steps, but comes closer to production deployment.
- c. Testing against a Docker Container using Docker/Testcontainers.
 - This involves very little Maven plugin configuration. Most of the work can be performed within the JUnit JVM.

Turn in one solution option, but if you do turn in multiple—identify which one is your primary option.

Do Not Make Use of ejava-parent Maven "it" Profile

For this assignment being primarily expressed within your provided solution project poms, do not make use of the Maven `it` profile activation in the `ejava-parent` pom. You may define your own `myit` Maven profile—with its own trigger in one of your intermediate parent poms—if you wish.

In short, this means do not declare the `it` Maven profile trigger file: `src/test/resources/application-it.properties`



Chapter 291. Assignment 4a: Executable JAR Option

291.1. Purpose

In this portion of the assignment, you will demonstrate your knowledge of implementing a Maven Failsafe integration test with a Spring Boot executable JAR. You will:

1. dynamically generate test instance-specific property values in order to be able to run in a shared CI/CD environment
2. start server process(es) configured with test instance-specific property values
3. implement a Maven Failsafe integration test in order to test a Spring Boot executable JAR running as a stand-alone process
4. execute tests against server process(es)
5. stop server process(es) at the conclusion of a test
6. evaluate test results once server process(es) have been stopped

291.2. Overview

In this portion of the assignment, you will be again testing your assignment 3 solution. However, this time as a stand-alone Spring Boot executable JAR running separately from the JUnit test.

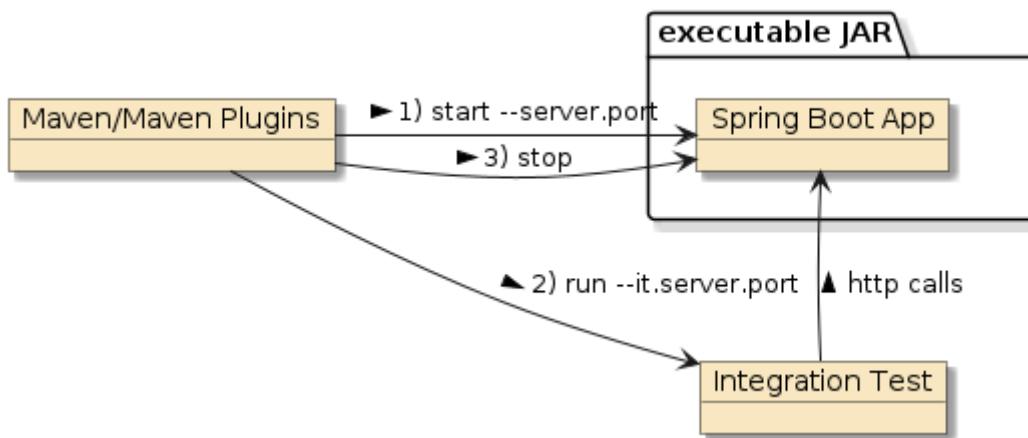


Figure 130. Maven Failsafe Integration Test

291.3. Requirements

1. Provide a Maven project pom that will dynamically generate a unique network server port prior to the Maven `pre-integration-test` phase.



The following shows an example output of satisfying the requirement using the Build Helper Maven Plugin and assigning it to Maven property `my.server.port`.

```
$ mvn verify  
...  
[INFO] --- build-helper:3.6.0:reserve-network-port (generate-port-number) @ autorentals-it-springboot ---  
[INFO] Reserved port 58262 for my.server.port
```

2. Provide a Maven project pom that will

- a. build the Spring Boot Executable JAR

The following shows an example output of satisfying the build requirement



```
[INFO] --- spring-boot-maven-plugin:3.3.2:repackage (build-app) @ autorentals-it-springboot ---  
[INFO] Creating repackaged archive .../target/autorentals-it-springboot-1.0-SNAPSHOT-bootexec.jar with classifier bootexec
```

- b. start the application as a Spring Boot executable JAR during the Maven **pre-integration-test** phase.

- i. The application API must listen to HTTP requests on the dynamically generated network server port

The following shows an example output of satisfying the requirement for start and assigning the port number.



```
[INFO] --- spring-boot-maven-plugin:3.3.2:start (start-server) @ autorentals-it-springboot ---  
...  
INFO 97896 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port 58262 (http)
```

- c. stop the application during the **post-integration-test** phase

The following shows an example output of satisfying the requirement for stopping the application.



```
[INFO] --- spring-boot-maven-plugin:3.3.2:stop (stop-server) @ autorentals-it-springboot ---  
[INFO] Stopping application...
```

3. Define a JUnit test that will test a call to a server-side component via an HTTP endpoint. This only has to be a single call. It does not matter what the call is as long as it is to something you developed, and it returns a success status. Security activation is **not required**. The call can be anonymous. There is **no requirement** to use injection or a Spring context on the client-side

even though the tips here show `@Bean` factories.

a. Accept/use a dynamically generated server port at runtime

The following shows an example of how the class examples have mapped "it.server.*" properties to a `@ConfigurationProperties` class. This can be leveraged to identify the base URL to the external server.

```
@Bean  
@ConfigurationProperties("it.server")  
ServerConfig serverConfig() {  
    return new ServerConfig();  
}
```



The following shows an example URL construction based upon the `@ConfigurationProperties` bean that has had server address information mapped to "it.server.*" properties.

```
@Bean  
URI rentalsURL(ServerConfig serverConfig) {  
    return UriComponentsBuilder.fromUri(serverConfig  
        .getBaseUrl()) ...
```

The following shows an example output of configuring JUnit to use the same dynamically assigned port number.

```
16:16:45.771 main DEBUG  
i.e.e.c.web.RestTemplateLoggingFilter#intercept:37  
GET http://localhost:58262/api/autorentals, returned 200 OK/200  
...  
{"contents":[]}
```

b. Not host or use any server-side components locally

To be fully compliant with this requirement, your JUnit test cannot use your `@SpringBootApplication` configuration of your application class. You should create a `@SpringBootConfiguration` as a substitute. Be sure to add `@EnableAutoConfiguration` if you need any automatically provided beans in the client (e.g., `RestTemplateBuilder`).



```
@SpringBootConfiguration  
@EnableAutoConfiguration  
public class IntegrationTestConfiguration {
```

4. Extend the Maven project pom to execute the JUnit test during the Maven `integration-test`

phase and ignored during the Maven **test** phase.



Surefire and Failsafe look for unique class naming patterns. Name the test class according to the phase it needs to execute in.



The Failsafe plugin must be explicitly configured to run the integration-test goal.

The following shows example output of Failsafe running tests.



```
[INFO] T E S T S
[INFO] -----
[INFO] Running
info.ejava_student.starter.assignment4.it.MyIntegrationIT
...
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time
elapsed: 1.975 s -- in
info.ejava_student.starter.assignment4.it.MyIntegrationIT
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

5. Extend the Maven project pom to evaluate test results once server process(es) have been stopped



The Failsafe plugin must be explicitly configured to run the verify goal.



The following shows example output of Failsafe verifying test results.

```
[INFO] --- maven-failsafe-plugin:3.3.1:verify (verify) @
autorentals-it-springboot ---
```

6. Turn in a source tree with complete Maven modules that will build and automatically test the application.

- a. Try to configure DEBUG so that the URL exchange is easy and obvious to observe.

The following shows example output of DEBUG with the full URL requested and the response.



```
16:16:45.771 main DEBUG
i.e.e.c.web.RestTemplateLoggingFilter#intercept:37
GET http://localhost:58262/api/autorentals, returned 200 OK/200
...
```

```
{"contents":[]}
```

291.3.1. Grading

Your solution will be evaluated on:

1. dynamically generate test instance-specific property values in order to be able to run in a shared CI/CD environment
 - a. whether the HTTP server port used between the JUnit test and the server-side components was dynamically generated such that each execution of the build results in a different server port number
2. start server process(es) configured with test instance-specific property values
 - a. whether the Spring Boot executable JAR was started with and uses a dynamically generated server port number
3. implement a Maven Failsafe integration test in order to test a Spring Boot executable JAR running as a stand-alone process
 - a. whether a JUnit test was provided that implements tests using HTTP requests
4. execute tests against server process(es)
 - a. whether the JUnit test is client-side-only, with server-side components deployed only to the remote application
5. stop server process(es) at the conclusion of a test
 - a. whether the started server process is terminated at the end of the overall tests
6. evaluate test results once server process(es) have been stopped
 - a. whether test results are actually evaluated and expressed in the overall Maven build result.

291.3.2. Additional Details

- You are permitted to implement the integration test in the same module or a downstream module relative to the actual Spring Boot application. However, there will be some extra issues related to sharing the Java JAR and repackaging into a Spring Boot Executable JAR in a downstream module to address.
 - the following shows an example downstream Integration Test module with only the IT artifacts and no **src/main** artifacts.

```
|-- pom.xml
\-- src
    |-- main
    |   '-- java (nothing)
    '-- test
        |-- java
        |   '-- info
    ...
    |       '-- it
```

```

|           |-- IntegrationTestConfiguration.java
|           '-- MyIntegrationIT.java
`-- resources
    '-- application-test.properties

```

- the downstream Integration Test module will have to declare a dependency on the upstream application module.

```

...
<dependency>
    <groupId>info.ejava-student.starter.assignment3.autorentals</groupId>
    <artifactId>autorentals-security-svc</artifactId>
    <version>1.0-SNAPSHOT</version>
</dependency>
...

```

- the downstream module can use the `@SpringBootApplication` of an upstream module when present in the upstream module dependency. The example below shows an explicit `mainClass` being identified in the upstream dependency.

```

...
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <configuration>

        <mainClass>info.ejava_student.starter.assignment3.security.AutoRentalsSecurityAp
p</mainClass>
    ...

```

- when running the IT test in the downstream module, the upstream application is what its being run. The following shows an example output from the upstream security application in the downstream IT module.

```

$ mvn spring-boot:run
...
INFO 6781 --- [ main] i.e.s.a.security.AutoRentalsSecurityApp : Starting
AutoRentalsSecurityApp using Java 17.0.12 with PID 6781
(.m2/repository/.../autorentals-security-svc-1.0-SNAPSHOT.jar started by jim in
.../assignment4-autorentals-it/autorentals-it-springboot)
...
INFO 6781 --- [ main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started
on port 8080 (http) with context path '/'
INFO 6781 --- [ main] i.e.s.a.security.AutoRentalsSecurityApp : Started
AutoRentalsSecurityApp in 1.782 seconds (process running for 2.064)

```

- You may want to temporarily force a JUnit assertion error to verify the test results are being verified by Maven at the end.

Chapter 292. Assignment 4b: Docker Plugin Option

There are two sections to this assignment option. You must complete them both.

292.1. Docker Image

292.1.1. Purpose

In this portion of the assignment, you will demonstrate your knowledge of building a Docker Image. You will:

1. build a layered Spring Boot Docker image using a Dockerfile and docker commands

292.1.2. Overview

In this portion of the assignment, you will be building a Docker image with your Spring Boot application organized in layers

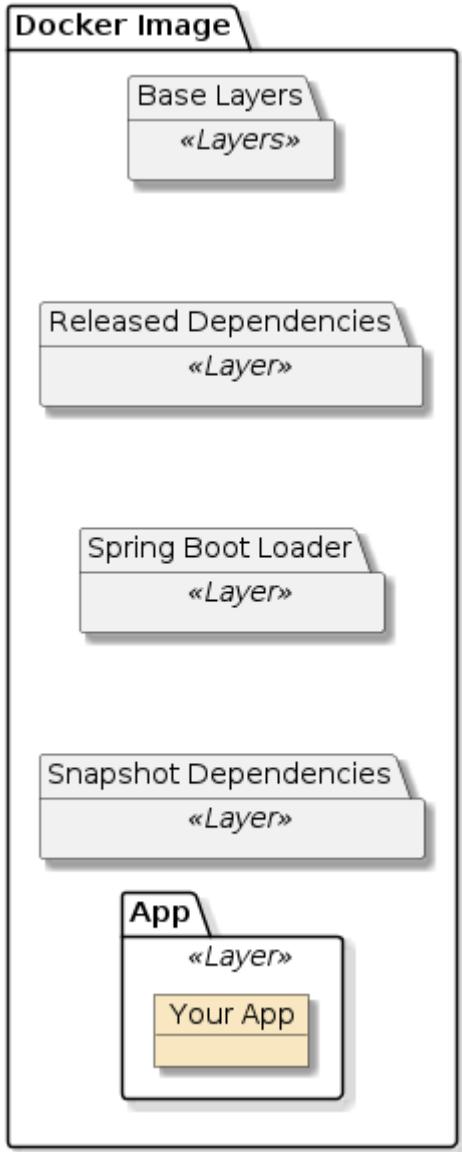


Figure 131. Layered Docker Image

292.1.3. Requirements

1. Provide a Maven project pom that will build the application as a Spring Boot executable JAR with the ability to extract layers.

Build Spring Boot Executable JAR

The following shows an example output of satisfying the requirement to build the application.



```
$ mvn clean package
...
[INFO] --- spring-boot-maven-plugin:3.3.2:repackage (build-app) @
autorentals-it-docker ---
[INFO] Creating repackaged archive .../target/autorentals-it-docker-
1.0-SNAPSHOT-bootexec.jar with classifier bootexec
...
```

Spring Boot Executable JAR is Layered

The following shows an example output verifying the application packaging supports layers.



```
$ java -Djarmode=tools -jar target/autorentals-it-docker-*-
bootexec.jar list-layers
dependencies
spring-boot-loader
snapshot-dependencies
application
```

2. Create a layered Docker image using a Dockerfile [multi-stage build](#) that will extend a JRE image and complete the image with your Spring Boot Application broken into separate layers.

Build Docker Image using Layers

The following shows an example output building a multi-stage Docker image, using layers.



```
$ docker build . -t test:test ①
[+] Building 5.3s (14/14) FINISHED
...
=> [builder 4/4] RUN java -Djarmode=tools -jar application.jar
extract --layers --launcher --destination extracted
...
=> [stage-1 3/6] COPY --from=builder
/builder/extracted/dependencies/ ./
=> [stage-1 4/6] COPY --from=builder /builder/extracted/spring-
boot-loader/ ./
=> [stage-1 5/6] COPY --from=builder /builder/extracted/snapshot-
dependencies/ ./
=> [stage-1 6/6] COPY --from=builder
/builder/extracted/application/ ./
=> => naming to docker.io/test:test
```

① the path provided will depend on the location you place your Dockerfile and files that go into the container

- a. The application API must listen to HTTP requests on a configured server port (with a default of 8080)
- b. The application API must activate with a provided profile
- c. The application should use HTTP and not HTTPS within the container.



Application Starts with Supplied Listen Port and Profile(s)

The following shows an example output of satisfying that the application can use a provided HTTP port number and profile(s).

```
$ docker run --rm test:test --server.port=7777  
--spring.profiles.active=nosecurity  
...  
AutoRentalsSecurityApp : The following 1 profile is active:  
"nosecurity"  
...  
TomcatWebServer : Tomcat started on port 7777 (http) with context  
path '/'
```

3. Turn in a source tree with complete Maven module containing the Dockerfile and source files. Use this exact same Dockerfile and application in the next section.

292.1.4. Grading

Your solution will be evaluated on:

1. build a layered Spring Boot Docker image using a Dockerfile and docker commands
 - a. whether you have a multi-stage Dockerfile
 - b. whether the Dockerfile successfully builds a layered version of your application using standard docker commands

292.1.5. Additional Details

- You may optionally choose to build and manage Dockerfile/Docker Compose tasks using the [io.fabric8:docker-maven-plugin](#), [io.brachu:docker-compose-maven-plugin](#), or other plugins.

292.2. Docker Integration Test

292.2.1. Purpose

In this portion of the assignment, you will demonstrate your knowledge of implementing a Maven Failsafe integration test using Docker or Docker Compose. You will:

1. dynamically generate test instance-specific property values in order to be able to run in a shared CI/CD environment
2. start server process(es) configured with test instance-specific property values
3. implement a Maven Failsafe integration test in order to test a Spring Boot executable JAR running within a Docker container as a stand-alone process
4. execute tests against server process(es)
5. stop server process(es) at the conclusion of a test
6. evaluate test results once server process(es) have been stopped

292.2.2. Overview

In this portion of the assignment you will be using a Docker container as your server process and managing it with Maven plugins.

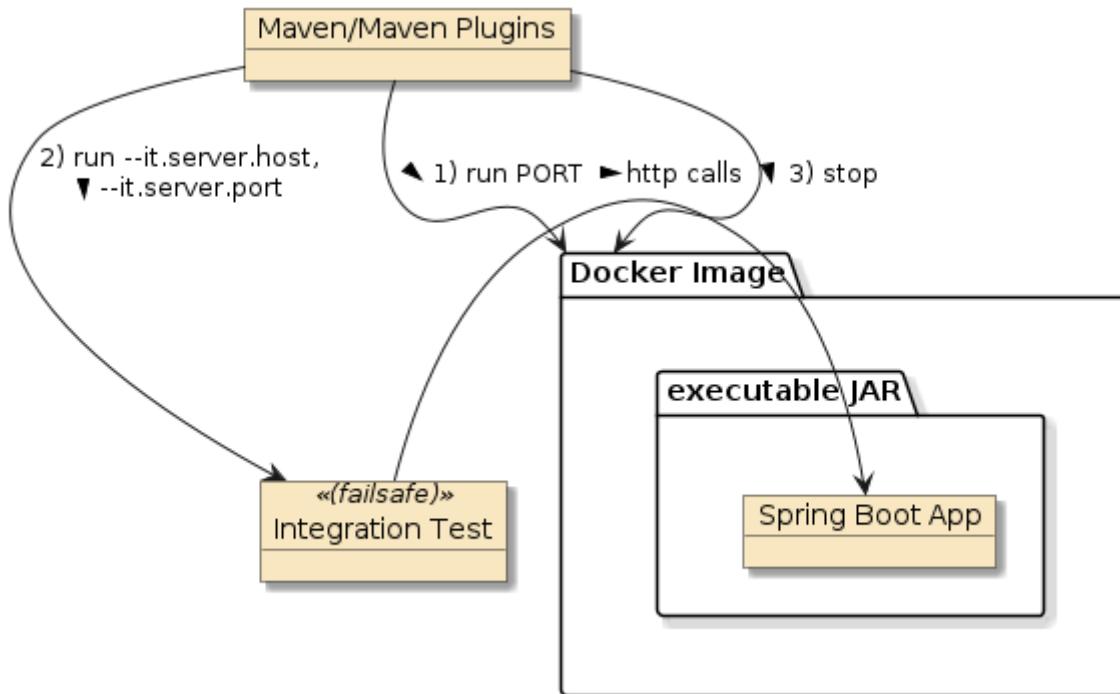


Figure 132. Spring Boot Docker Integration Test

292.2.3. Requirements

1. Provide a Maven project pom that will build the Spring Boot executable JAR (from the previous section)
2. Provide a Maven project pom that will dynamically generate a unique network server port prior to the Maven **pre-integration-test** phase.

Generate Unique Network Port Number

The following shows an example output of satisfying the port allocation requirement using the Build Helper Maven Plugin and assigning it to Maven property `my.server.port`.



```
$ mvn verify  
...  
[INFO] --- build-helper-maven-plugin:3.6.0:reserve-network-port  
(generate-port-number) @ autorentals-it-docker ---  
[INFO] Reserved port 52368 for my.server.port
```

3. Provide a Maven project pom that will build the Docker image prior to the **pre-integration-test** phase.



If your Dockerfile is below the `${project.basedir}`, you may have to provide a `contextDir` setting to `${project.basedir}` to indicate where all relatives paths

start.

```
<docker.contextDir>${project.basedir}</docker.contextDir>
```

Build Docker Image within Project Build

The following shows an example output of satisfying the Docker build requirement using the `io.fabric8:docker-maven-plugin` Maven plugin.



```
$ mvn package
...
[INFO] --- docker-maven-plugin:0.44.0:build (build-image) @
autorentals-it-docker ---
[INFO] Building tar: .../target/docker/autorentals-it-docker/1.0-
SNAPSHOT/tmp/docker-build.tar
[INFO] DOCKER> [autorentals-it-docker:1.0-SNAPSHOT] "autorentals-it-
docker": Created docker-build.tar in 483 milliseconds
[INFO] DOCKER> Step 1/14 : FROM eclipse-temurin:17-jre AS builder
....
[INFO] DOCKER> Successfully built 04d5a32d0c34
[INFO] DOCKER> Successfully tagged autorentals-it-docker:1.0-
SNAPSHOT
[INFO] DOCKER> [autorentals-it-docker:1.0-SNAPSHOT] "autorentals-it-
docker": Built image sha256:04d5a
[INFO] DOCKER> autorentals-it-docker:1.0-SNAPSHOT: Removed dangling
image sha256:fbcfd3
```

4. Provide a Maven project pom that will start the application within a Docker image/container during the Maven `pre-integration-test` phase. You may use straight Docker or Docker Compose.



Start Docker Container Prior to Integration Test

The following shows an example output of satisfying the Docker start requirement using the `io.fabric8:docker-maven-plugin` Maven plugin.

```
$ mvn pre-integration-test
...
[INFO] --- docker-maven-plugin:0.44.0:start (start-container) @
autorentals-it-docker ---
[INFO] DOCKER> [autorentals-it-docker:1.0-SNAPSHOT] "autorentals-it-
docker": Start container e95c730579be
[INFO] DOCKER> [autorentals-it-docker:1.0-SNAPSHOT] "autorentals-it-
docker": Waiting on url http://localhost:52368/api/autorentals with
method HEAD for status 200..399.
[INFO] DOCKER> [autorentals-it-docker:1.0-SNAPSHOT] "autorentals-it-
docker": Waited on url http://localhost:52368/api/autorentals 3321
ms=
```

Verify Container is Running and Mapped to Dynamic Port

The following output demonstrates checking the container is running and allocated port number assigned.



```
$ docker ps
CONTAINER ID   IMAGE
1214c08d9732   autorentals-it-docker:1.0-SNAPSHOT
                >8080/tcp
PORTS          0.0.0.0:52368-
```

Stop Container



The following demonstrates how to stop the container(s) associated with this module using the [io.fabric8:docker-maven-plugin](#) Maven plugin.

```
$ mvn docker:stop
```

- a. The application's `server.port` must be mapped to the dynamically generated network server port for the running Docker container
 - b. The application must use a provided Spring profile (e.g., `nosecurity`)
 - c. The application may use HTTP (not HTTPS)
5. Define a JUnit test that will test the server-side components via HTTP endpoints
 - a. Accept/use a dynamically generated server port at runtime
 - b. Not host or use any server-side components locally
 6. Extend the Maven project pom to execute the JUnit test during the Maven `integration-test` phase and ignored during the Maven `test` phase.

Execute JUnit Integration Test with Failsafe



The following shows an example output of satisfying the JUnit execution requirement using the Failsafe Maven Plugin.

```
$ mvn verify
...
[INFO] --- maven-failsafe-plugin:3.3.1:integration-test
(integration-test) @ autorentals-it-docker ---
...
[INFO] Running
info.ejava_student.starter.assignment4.it.MyIntegrationIT
...
17:16:15.678 main  INFO
.e.s.assignment4.it.MyIntegrationIT#logStarting:50 Starting
MyIntegrationIT using Java 17.0.12 with PID 18304 (started in
.../autorentals-it-docker)
...
17:16:17.284 main DEBUG
```

```
i.e.e.c.web.RestTemplateLoggingFilter#intercept:37
GET http://localhost:52429/api/autorentals, returned 200 OK/200
sent: [Accept:"text/plain, application/json, ...

rcvd: [Vary:"Origin", "Access-Control-Request-Method", "Access-
Control-Request-Headers", ...
>{"contents":[]}
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time
elapsed: 2.366 s -- in
info.ejava_student.starter.assignment4.it.MyIntegrationIT
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

7. Extend the Maven project pom to stop the Docker container once the test(s) have completed

Stop Docker Container After Integration Tests Complete

The following shows an example output of satisfying the Docker stop requirement using the `io.fabric8:docker-maven-plugin` Maven plugin.



```
$ mvn verify
...
[INFO] --- docker-maven-plugin:0.44.0:stop (stop-container) @
autorentals-it-docker ---
[INFO] DOCKER> [autorentals-it-docker:1.0-SNAPSHOT] "autorentals-it-
docker": Stop and removed container 1214c08d9732 after 0 ms
```

8. Extend the Maven project pom to evaluate test results once server process(es) have been stopped



Verify Integration Test Results After Resources have Stopped

```
[INFO] --- maven-failsafe-plugin:3.3.1:verify (verify) @
autorentals-it-docker ---
[INFO]
-----
[INFO] BUILD SUCCESS
```

9. Turn in a source tree with complete Maven modules that will build and automatically test the application.

- a. Try to configure DEBUG so that the URL exchange is easy and obvious to observe.



The following shows example output of DEBUG with the full URL requested and the response.

```
16:16:45.771 main DEBUG
i.e.e.c.web.RestTemplateLoggingFilter#intercept:37
GET http://localhost:52429/api/autorentals, returned 200 OK/200
...
{"contents":[]}
```

292.2.4. Grading

Your solution will be evaluated on:

1. dynamically generate test instance-specific property values in order to be able to run in a shared CI/CD environment
 - a. whether the HTTP server port used between the JUnit test and the server-side components was dynamically generated such that each execution of the build results in a different server port number used on the Docker host
2. start server process(es) configured with test instance-specific property values
 - a. whether the Docker container hosting the Spring Boot executable JAR was started and had its server port mapped to the dynamically generated value
3. implement a Maven Failsafe integration test in order to test a Spring Boot executable JAR running within a Docker container as a stand-alone process
 - a. whether a JUnit test was provided that implements tests using HTTP requests
4. execute tests against server process(es)
 - a. whether the JUnit test is client-side-only, with server-side components deployed only to the remote application
5. stop server process(es) at the conclusion of a test
 - a. whether the started server process is terminated at the end of the overall tests
6. evaluate test results once server process(es) have been stopped
 - a. whether test results are actually evaluated and expressed in the overall Maven build result.

292.2.5. Additional Details

- You are permitted to implement the integration test in the same module or a downstream module relative to the actual Spring Boot application. However, there will be some extra issues related to sharing the Java JAR and repackaging into a Spring Boot Executable JAR in a downstream module to address.
 - the following shows an example downstream Integration Test module with only the IT artifacts and no **src/main** artifacts.

```
-- pom.xml
`-- src
    |-- main
        |-- docker ①
```

```

    |   |   '-- Dockerfile
    |   '-- java (nothing)
    '-- test
        |-- java
            '-- info
...
        '-- it
            |-- IntegrationTestConfiguration.java
            '-- MyIntegrationIT.java
    '-- resources
        '-- application-test.properties

```

- ① the Dockerfile can be anywhere in your tree. This directory would likely house any additional artifacts that are part of the image.
- the downstream Integration Test module will have to declare a dependency on the upstream application module.

```

...
<dependency>
    <groupId>info.ejava-student.starter.assignment3.autorentals</groupId>
    <artifactId>autorentals-security-svc</artifactId>
    <version>1.0-SNAPSHOT</version>
</dependency>
...

```

- the downstream module can use the `@SpringBootApplication` of an upstream module when present in the upstream module dependency. The example below shows an explicit `mainClass` being identified in the upstream dependency.

```

...
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <configuration>

        <mainClass>info.ejava_student.starter.assignment3.security.AutoRentalsSecurityAp
p</mainClass>
    ...

```

- when running the IT test in the downstream module, the upstream application is what its being run. The following shows an example output from the upstream security application in the downstream IT module.

```

$ mvn spring-boot:run
...
INFO 6781 --- [ main] i.e.s.a.security.AutoRentalsSecurityApp : Starting
AutoRentalsSecurityApp using Java 17.0.12 with PID 6781

```

```
(.m2/repository/.../autorentals-security-svc-1.0-SNAPSHOT.jar started by jim in  
.../assignment4-autorentals-it/autorentals-it-springboot)  
...  
INFO 6781 --- [ main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started  
on port 8080 (http) with context path '/'  
INFO 6781 --- [ main] i.e.s.a.security.AutoRentalsSecurityApp : Started  
AutoRentalsSecurityApp in 1.782 seconds (process running for 2.064)
```

- You may want to temporarily force a JUnit assertion error to verify the test results are being verified by Maven at the end.
- You may optionally choose to build and manage Dockerfile/Docker Compose tasks using the [io.fabric8:docker-maven-plugin](#), [io.brachu:docker-compose-maven-plugin](#), or other plugins.
- `localhost` is the Docker host during normal development. That means JUnit can locate the Docker container using `localhost` during development. However, `localhost` is not the Docker host in most Docker CI/CD environments. To be portable, your JUnit test must obtain the hostname from at runtime. The value ends up commonly `host.docker.internal` and is discussed towards the end of the Docker IT lecture.

Chapter 293. Assignment 4c: Testcontainers Option

There are two sections to this assignment option. You must complete them both.

293.1. Docker Image

293.1.1. Purpose

In this portion of the assignment, you will demonstrate your knowledge of building a Docker Image. You will:

1. build a layered Spring Boot Docker image using a Dockerfile and docker commands

293.1.2. Overview

In this portion of the assignment, you will be building a Docker image with your Spring Boot application organized in layers

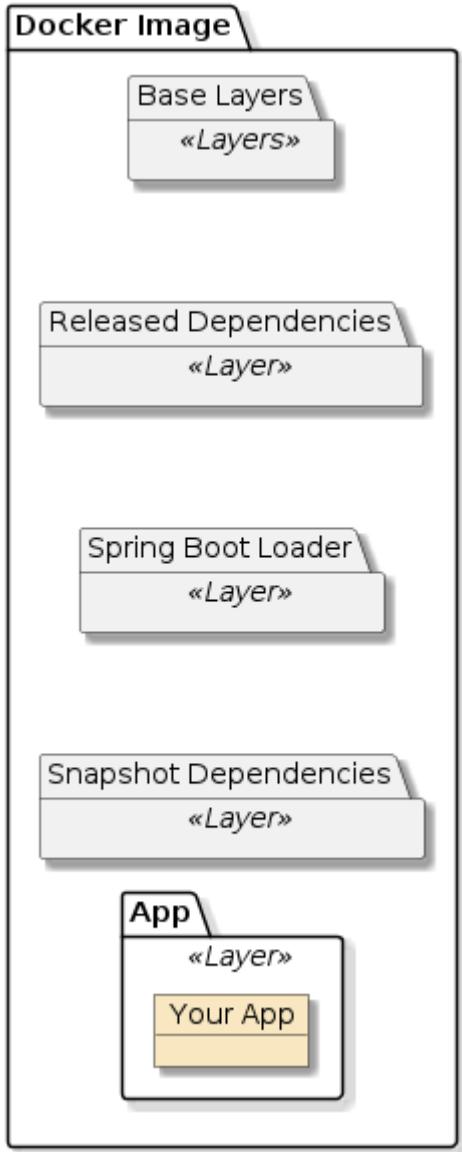


Figure 133. Layered Docker Image

293.1.3. Requirements

1. Provide a Maven project pom that will build the application as a Spring Boot executable JAR with the ability to extract layers.

Build Spring Boot Executable JAR

The following shows an example output of satisfying the requirement to build the application.



```
$ mvn clean package
...
[INFO] --- spring-boot-maven-plugin:3.3.2:repackage (build-app) @
autorentals-it-docker ---
[INFO] Creating repackaged archive .../target/autorentals-it-docker-
1.0-SNAPSHOT-bootexec.jar with classifier bootexec
...
```

Spring Boot Executable JAR is Layered

The following shows an example output verifying the application packaging supports layers.



```
$ java -Djarmode=tools -jar target/autorentals-it-docker-*-
bootexec.jar list-layers
dependencies
spring-boot-loader
snapshot-dependencies
application
```

2. Create a layered Docker image using a Dockerfile [multi-stage build](#) that will extend a JRE image and complete the image with your Spring Boot Application broken into separate layers.

Build Docker Image using Layers

The following shows an example output building a multi-stage Docker image, using layers.



```
$ docker build . -t test:test ①
[+] Building 5.3s (14/14) FINISHED
...
=> [builder 4/4] RUN java -Djarmode=tools -jar application.jar
extract --layers --launcher --destination extracted
...
=> [stage-1 3/6] COPY --from=builder
/builder/extracted/dependencies/ ./
=> [stage-1 4/6] COPY --from=builder /builder/extracted/spring-
boot-loader/ ./
=> [stage-1 5/6] COPY --from=builder /builder/extracted/snapshot-
dependencies/ ./
=> [stage-1 6/6] COPY --from=builder
/builder/extracted/application/ ./
=> => naming to docker.io/test:test
```

① the path provided will depend on the location you place your Dockerfile and files that go into the container

- a. The application API must listen to HTTP requests on a configured server port (with a default of 8080)
- b. The application API must activate with a provided profile
- c. The application should use HTTP and not HTTPS within the container.



Application Starts with Supplied Listen Port and Profile(s)

The following shows an example output of satisfying that the application can use a provided HTTP port number and profile(s).

```
$ docker run --rm test:test --server.port=7777  
--spring.profiles.active=nosecurity  
...  
AutoRentalsSecurityApp : The following 1 profile is active:  
"nosecurity"  
...  
TomcatWebServer : Tomcat started on port 7777 (http) with context  
path '/'
```

3. Turn in a source tree with complete Maven module containing the Dockerfile and source files.
Use this exact same Dockerfile and application in the next section.

293.1.4. Grading

Your solution will be evaluated on:

1. build a layered Spring Boot Docker image using a Dockerfile and docker commands
 - a. whether you have a multi-stage Dockerfile
 - b. whether the Dockerfile successfully builds a layered version of your application using standard docker commands

293.1.5. Additional Details

- You may optionally choose to build and manage Dockerfile/Docker Compose tasks using the [io.fabric8:docker-maven-plugin](#), [io.brachu:docker-compose-maven-plugin](#), or other plugins.

293.2. Testcontainers Integration Test

293.2.1. Purpose

In this portion of the assignment, you will demonstrate your knowledge of implementing a Maven Failsafe integration test using Docker and Testcontainers. You will:

1. start server container(s) prior to the start of tests
2. implement a Maven Failsafe integration test in order to test a Spring Boot executable JAR running within a Docker container as a stand-alone process
3. execute tests against server container(s)
4. stop server container(s) at the conclusion of tests
5. evaluate test results once server process(es) have been stopped

293.2.2. Overview

In this portion of the assignment you will be using a Docker container as your server process and managing it with Testcontainers.

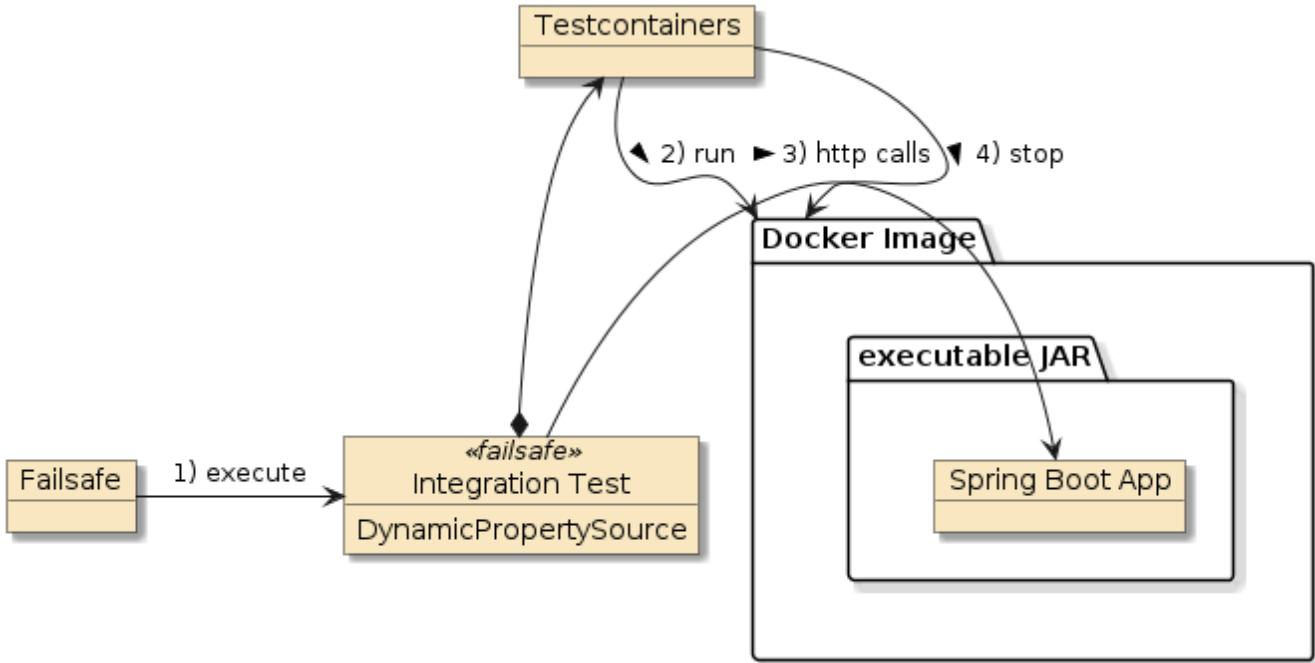


Figure 134. Spring Boot Testcontainers Integration Test

293.2.3. Requirements

1. Provide a Maven project pom that will build the Spring Boot executable JAR (from the previous section)
2. Augment the project pom with Testcontainers dependencies for base capability and integration with JUnit.
3. Provide a JUnit test that will get executed during the **integration-test** phase (after the Spring Boot Executable JAR has been created) by the Maven Failsafe plugin.

Vanilla Failsafe Test



There are no inputs to the test. It just has an ordering requirement to come after the Spring Boot executable JAR has been built.

4. The JUnit test shall:

- a. create a Testcontainers-managed container for the Spring Boot executable JAR using either:

This means that you only define the container. Testcontainers will start and stop and provide information about the container.



- `@Testcontainers`
- `@Container`

- i. source Dockerfile (best and easiest)

See [Creating Images on-the-fly](#).



- `new ImageFromDockerfile()`

- `withFileFromPath()` - for both Dockerfile and contextPath
- ii. an existing Docker image pre-built by the project's pom using Maven plugins
 - b. expose the default API 8080 port
 - c. run the application with the desired Spring profile
-  See [Other Common Property Sources](#) course notes on options you have to set the `profile` property.
- d. Dynamically assign any JUnit test-required properties that are derived from the running container.
- 
- `it.server.host`
 - `it.server.port`
5. Turn in a source tree with complete Maven modules that will build and automatically test the application.

The following is example output for a Maven/Failsafe build executing a JUnit test using Testcontainers



```
$ mvn clean verify
...
[INFO] --- spring-boot-maven-plugin:3.3.2:repackage (build-app) @ autorentals-it-testcontainers ---
[INFO] Creating repackaged archive .../target/autorentals-it-
testcontainers-1.0-SNAPSHOT-bootexec.jar with classifier bootexec
...
[INFO] --- maven-failsafe-plugin:3.3.1:integration-test
(integration-test) @ autorentals-it-testcontainers ---
[INFO] Running
info.ejava_student.starter.assignment4.it.MyTestContainersIT
...
INFO tc.localhost/testcontainers/fcbqgu1j8yyesl3b:latest -- Creating
container for image:
localhost/testcontainers/fcbqgu1j8yyesl3b:latest
...
DEBUG i.e.e.c.web.RestTemplateLoggingFilter#intercept:37
GET http://localhost:56185/api/autorentals, returned 200 OK/200
sent: [Accept:"text/plain, application/json, application/xml,
text/xml, application/*+json, application/*+xml, */*", Content-
Length:"0"]

rcvd: [Vary:"Origin", "Access-Control-Request-Method", "Access-
Control-Request-Headers", X-Content-Type-Options:"nosniff", X-XSS-
Protection:"0", Cache-Control:"no-cache, no-store, max-age=0, must-
revalidate", Pragma:"no-cache", Expires:"0", X-Frame-Options:"DENY",
Content-Type:"application/json", Transfer-Encoding:"chunked",
```

```
Date:"Sat, 12 Oct 2024 15:14:07 GMT", Keep-Alive:"timeout=60",
Connection:"keep-alive"]
{"contents":[]}
```

```
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time
elapsed: 15.86 s -- in
info.ejava_student.starter.assignment4.it.MyTestContainersIT
```

293.2.4. Grading

Your solution will be evaluated on:

1. implement a Maven Failsafe integration test in order to test a Spring Boot executable JAR running within a Docker container as a stand-alone process
 - a. whether the container was automatically started/stopped by Testcontainers
 - b. whether the Docker image was built within the scope of this module (either by Testcontainers during the test or via Maven plugins prior to the test)
 - c. whether the host and port for the container is obtained from Testcontainers
 - d. whether a JUnit test was provided that implements tests using HTTP requests
2. execute tests against server process(es)
 - a. whether the JUnit test is client-side-only, with server-side components deployed only to the remote application
3. evaluate test results once server process(es) have been stopped
 - a. whether test results are actually evaluated and expressed in the overall Maven build result.

293.2.5. Additional Details

- You are permitted to implement the integration test in the same module or a downstream module relative to the actual Spring Boot application. However, there will be some extra issues related to sharing the Java JAR and repackaging into a Spring Boot Executable JAR in a downstream module to address.
 - the following shows an example downstream Integration Test module with only the **IT** artifacts and no **src/main** artifacts.

```
 |-- pom.xml
`-- src
    |-- main
    |   |-- docker ①
    |   |   '-- Dockerfile
    |   '-- java (nothing)
    '-- test
        |-- java
        |   '-- info
...
...
```

```

    |   '-- it
    |       |-- IntegrationTestConfiguration.java
    |       '-- MyTestContainersIT.java
    '-- resources
        '-- application-test.properties

```

- ① the Dockerfile can be anywhere in your tree. This directory would likely house any additional artifacts that are part of the image.
- the downstream Integration Test module will have to declare a dependency on the upstream application module.

```

...
<dependency>
    <groupId>info.ejava-student.starter.assignment3.autorentals</groupId>
    <artifactId>autorentals-security-svc</artifactId>
    <version>1.0-SNAPSHOT</version>
</dependency>
...

```

- the downstream module can use the `@SpringBootApplication` of an upstream module when present in the upstream module dependency. The example below shows an explicit `mainClass` being identified in the upstream dependency.

```

...
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <configuration>

        <mainClass>info.ejava_student.starter.assignment3.security.AutoRentalsSecurityAp
p</mainClass>
    ...

```

- when running the IT test in the downstream module, the upstream application is what its being run. The following shows an example output from the upstream security application in the downstream IT module.

```

$ mvn spring-boot:run -Dspring-boot.run.profiles=nosecurity
...
INFO 6781 --- [ main] i.e.s.a.security.AutoRentalsSecurityApp : ...

Starting AutoRentalsSecurityApp using Java 17.0.12 with PID 34247
(.m2/repository/.../autorentals-security-svc-1.0-SNAPSHOT.jar started by jim in
.../assignment4-autorentals-it/autorentals-it-testcontainers)
...
INFO 6781 --- [ main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started
on port 8080 (http) with context path '/'

```

```
INFO 6781 --- [ main] i.e.s.a.security.AutoRentalsSecurityApp : The following 1
profile is active: "nosecurity"
...
INFO 6781 --- [ main] i.e.s.a.security.AutoRentalsSecurityApp : Started
AutoRentalsSecurityApp in 1.782 seconds (process running for 2.064)
```

- You may want to temporarily force a JUnit assertion error to verify the test results are being verified by Maven at the end.
- You may optionally choose to build the Docker image using Testcontainers and a Dockerfile. Alternately, you may chose to use [io.fabric8:docker-maven-plugin](#), [io.brachu:docker-compose-maven-plugin](#), or other Maven plugins outside of Testcontainers—but that would defeat much of the simplicity of this overall option.
- `localhost` is the Docker host during normal development. That means JUnit can locate the Docker container using `localhost` during development. However, `localhost` is not the Docker host in most Docker CI/CD environments. To be portable, your JUnit test must obtain the hostname from at runtime. The value ends up commonly `host.docker.internal` and is discussed towards the end of the Docker IT lecture.

RDBMS

copyright © 2024 jim stafford (jim.stafford@jhu.edu)

Chapter 294. Introduction

This lecture will introduce working with relational databases with Spring Boot. It includes the creation and migration of schema, SQL commands, and low-level application interaction with JDBC.

294.1. Goals

The student will learn:

- to identify key parts of a RDBMS schema
- to instantiate and migrate a database schema
- to automate database schema migration
- to interact with database tables and rows using SQL
- to identify key aspects of Java Database Connectivity (JDBC) API

294.2. Objectives

At the conclusion of this lecture and related exercises, the student will be able to:

1. define a database schema that maps a single class to a single table
2. implement a primary key for each row of a table
3. define constraints for rows in a table
4. define an index for a table
5. automate database schema migration with the Flyway tool
6. manipulate table rows using SQL commands
7. identify key aspects of a JDBC call

Chapter 295. Schema Concepts

Relational databases are based on a set of explicitly defined tables, columns, constraints, sequences, and indexes. The overall structure of these definitions is called **schema**. Our first example will be a single table with a few columns.

295.1. RDBMS Tables/Columns

A table is identified by a name and contains a flat set of fields called "columns". It is common for the table name to have an optional scoping prefix in the event that the database is shared, (e.g., during testing or a minimal deployment).

In this example, the `song` table is prefixed by a `reposongs_` name that identifies which course example this table belongs to.

Example Table and Columns

Table "public.reposongs_song" ①

Column	
id	②
title	③
artist	
released	

① table named `reposongs_song`, part of the `reposongs` schema

② column named `id`

③ column named `title`

295.2. Column Data

Individual tables represent a specific type of object and their columns hold the data. Each row of the `song` table will always have an `id`, `title`, `artist`, and `released` column.

Example Table/Column Data

id	title	artist	released
1	Noli Me Tangere	Orbital	2002-07-06
2	Moab Is My Washpot	Led Zeppelin	2005-03-26
3	Arms and the Man	Parliament Funkadelic	2019-03-11

295.3. Column Types

Each column is assigned a type that constrains the type and size of value they can hold.

Song Column Types

Table "public.reposongs_song"		
Column	Type	
id	integer	①
title	character varying(255)	②
artist	character varying(255)	
released	date	③

① `id` column has type integer

② `title` column has type varchar that is less than or equal to 255 characters

③ `released` column has type `date`

295.4. Example Column Types

The following lists several common example column data types. A more complete list of column types can be found on the [w3schools website](#). Some column types can be vendor-specific.

Table 19. Example Column Types

Category	Example Type
Character Data	<ul style="list-style-type: none">char(size) - a fixed length set of charactersvarchar(size) - a variable length of charactersblob(size), clob(size) - a large field of binary or textual data
Boolean/ Numeric data	<ul style="list-style-type: none">boolean - true/false valueint(size), bigint(size) - numeric valuenumeric(size, digits) and/or decimal(size, digits) - fixed-point number. e.g., money. numeric definition is more strict in size. decimal definition is "at least" in size. In practice — they tend to be the same.
Temporal data	<ul style="list-style-type: none">date - date without timetime - time without datedatetime - a specific time on a specific date. Timezone is commonly UTC

 *Character field maximum size is vendor-specific*

The maximum size of a char/varchar column is vendor-specific, ranging from 4000 characters to much larger values.

295.5. Constraints

Column values are constrained by their defined type and can be additionally constrained to be required (`not null`), unique (e.g., primary key), a valid reference to an existing row (foreign key), and various other constraints that will be part of the total schema definition.

The following example shows a required column and a unique primary key constraint.

Example Column Types

```
postgres=# \d reposongs_song
          Table "public.reposongs_song"
  Column |           Type            | Nullable |
-----+-----+-----+
  id    | integer             | not null | ①
 title | character varying(255) |           |
 artist | character varying(255) |           |
 released | date            |           |
Indexes:
"song_pk" PRIMARY KEY, btree (id) ②
```

① column **id** is required

② column **id** constrained to hold a unique (primary) key for each row

295.6. Primary Key

A primary key is used to uniquely identify a specific row within a table and can also be the target of incoming references (foreign keys). There are two origins of a primary key: natural and surrogate. Natural primary keys are derived directly from the business properties of the object. Surrogate primary keys are externally generated and added to the business properties.

The following identifies the two primary key origins and lists a few advantages and disadvantages.

Table 20. Primary Key Origins

Primary Key Origins	Natural PKs	Surrogate PKs
Description	derived directly from business properties of an object	externally generated and added to object
Example	<ul style="list-style-type: none">employee ID (e123)e-mail address (me@gmail.com)	<ul style="list-style-type: none">centrally generated sequence number (1,2,3)distributed generated UUID (594075a4-5578-459f-9091-e7734d4f58ce)
Advantages	<ul style="list-style-type: none">no new fields are necessaryID can be determined before DB insert	<ul style="list-style-type: none">guaranteed to be uniqueunique business properties permitted to change (e.g. switch email address)

Primary Key Origins	Natural PKs	Surrogate PKs
Disadvantages	<ul style="list-style-type: none"> business properties for ID are each required business properties for ID cannot change sometimes requires combining multiple properties (i.e., "compound primary key")—which complicates foreign keys 	<ul style="list-style-type: none"> a new field must be added visible sequences can be guessed and deterministic increments can be used for size and rate measurement

For this example, I am using a surrogate primary key that could have been based on either a UUID or sequence number.

295.7. UUID

A UUID is a globally unique 128 bit value written in hexadecimal, broken up into five groups using dashes, resulting in a 36 character string.

Example UUID

```
$ uuidgen | awk '{print tolower($0)}'
594075a4-5578-459f-9091-e7734d4f58ce
```

There are different versions of the algorithm, but each target the same structure and the negligible chance of duplication.^[1] This provides not only a unique value for the table row, but also a unique value across all tables, services, and domains.

The following lists a few advantages and disadvantages for using UUIDs as a primary key.

Table 21. UUID as Primary Key

UUID Advantages	UUID Disadvantages
<ul style="list-style-type: none"> globally unique <ul style="list-style-type: none"> easier to search through logs containing information from many tables can be generated anytime and anywhere <ul style="list-style-type: none"> object does not have to wait to be inserted into DB before having an ID—feature similar to natural keys 	<ul style="list-style-type: none"> larger than needed to be unique for only a table <ul style="list-style-type: none"> requires more storage space slower to compare relative to a smaller integer value <ul style="list-style-type: none"> requires additional comparison time

295.8. Database Sequence

A database sequence is a numeric value guaranteed to be unique by the database. Support for sequences and the syntax used to work with them varies per database. The following shows an

example of creating, incrementing, and dropping a sequence in postgres.

postgres sequence value

```
postgres=# create sequence seq_a start 1 increment 1; ①
CREATE SEQUENCE

postgres=# select nextval('seq_a'); ②
nextval
-----
 1
(1 row)

postgres=# select nextval('seq_a');
nextval
-----
 2
(1 row)

postgres=# drop sequence seq_a;
DROP SEQUENCE
```

① can define starting point and increment for sequence

② obtain next value of sequence using a database query



Database Sequences do not dictate how unique value is used

Database Sequences do not dictate how the unique value is used. The caller can use that directly as the primary key for one or more tables or anything at all. The caller may also use the returned value to self-generate IDs on its own (e.g., a page offset of IDs). That is where the **increment** option can be of use.

295.8.1. Database Sequence with Increment

We can use the **increment** option to help maintain a 1:1 ratio between sequence and primary key values—while giving the caller the ability to self-generate values within a increment window.

Database Sequence with Increment

```
postgres=# create sequence seq_b start 1 increment 100; ①
CREATE SEQUENCE
postgres=# select nextval('seq_b');
nextval
-----
 1 ①
(1 row)

postgres=# select nextval('seq_b');
nextval
-----
```

101 ①
(1 row)

① increment leaves a window of values that can be self-generated by caller

The database client calls `nextval` whenever it starts or runs out of a window of IDs. This can cause gaps in the sequence of IDs.

Chapter 296. Example POJO

We will be using an example `Song` class to demonstrate some database schema and interaction concepts. Initially, I will only show the POJO portions of the class required to implement a business object and manually map this to the database. Later, I will add some JPA mapping constructs to automate the database mapping.

The class is a read-only value class with only constructors and getters. We cannot use the lombok `@Value` annotation because JPA (part of a follow-on example) will require us to define a no argument constructor and attributes cannot be final.

Song POJO being mapped to database

```
package info.ejava.examples.db.repo.jpa.songs.bo;  
...  
@Getter ①  
@ToString  
@Builder  
@AllArgsConstructor  
@NoArgsConstructor  
public class Song {  
    private int id; ②  
    private String title;  
    private String artist;  
    private LocalDate released;  
}
```

① each property will have a getter method() but the only way to set values is through the constructor.builder

② surrogate primary key will be used as a primary key

POJOs can be read/write



There is no underlying requirement to use a read-only POJO with JPA or any other mapping. However, doing so does make it more consistent with **DDD read-only entity** concepts where changes are through explicit save/update calls to the repository versus subtle side-effects of calling an entity `setter()`.

Chapter 297. Schema

To map this class to the database, we will need the following constructs:

- a table
- a sequence to generate unique values for primary keys
- an integer column to hold `id`
- 2 varchar columns to hold `title` and `artist`
- a date column to hold `released`

The constructs are defined by `schema`. Schema is instantiated using specific commands. Most core schema creation commands are vendor neutral. Some schema creation commands (e.g., `IF EXISTS`) and options are vendor-specific.

297.1. Schema Creation

Schema can be

- authored by hand,
- auto-generated, or
- a mixture of the two.

We will have the tooling necessary to implement auto-generation once we get to JPA, but we are not there yet. For now, we will start by creating a complete schema definition by hand.

297.2. Example Schema

The following example defines a sequence and a table in our database ready for use with postgres.

Schema Creation Example (V1.0.0_initial_schema.sql)

```
drop sequence IF EXISTS reposongs_song_sequence; ①
drop table IF EXISTS reposongs_song;

create sequence reposongs_song_sequence start 1 increment 50; ②
create table reposongs_song (
    id int not null,
    title varchar(255),
    artist varchar(255),
    released date,
    constraint song_pk primary key (id)
);

comment on table reposongs_song is 'song database'; ③
comment on column reposongs_song.id is 'song primary key';
comment on column reposongs_song.title is 'official song name';
```

```
comment on column reposongs_song.artist is 'who recorded song';
comment on column reposongs_song.released is 'date song released';

create index idx_song_title on reposongs_song(title);
```

- ① remove any existing residue
- ② create new DB table(s) and sequence
- ③ add descriptive comments

Chapter 298. Schema Command Line Population

To instantiate the schema, we have the option to use the command line interface (CLI). The following example connects to a database running within docker-compose. The `psql` CLI is executed on the same machine as the database, thus saving us the requirement of supplying the password. The contents of the schema file is supplied as `stdin`.

Schema Command Line Population

```
$ docker-compose up -d postgres
Creating ejava_postgres_1 ... done

$ docker-compose exec -T postgres psql -U postgres \① ②
< .../src/main/resources/db/migration/V1.0.0_0__initial_schema.sql ③
DROP SEQUENCE
DROP TABLE
NOTICE: sequence "reposongs_song_sequence" does not exist, skipping
NOTICE: table "reposongs_song" does not exist, skipping
CREATE SEQUENCE
CREATE TABLE
COMMENT
COMMENT
COMMENT
COMMENT
COMMENT
```

① running `psql` CLI command on `postgres` image

② `-T` disables docker-compose pseudo-tty allocation

③ reference to schema file on host



Pass file using stdin

The file is passed in through `stdin` using the "`<`" character. Do not miss adding the "`<`" character.

The following schema commands add an index to the `title` field.

Additional Schema Index

```
$ docker-compose exec -T postgres psql -U postgres \
< .../src/main/resources/db/migration/V1.0.0_1__initial_indexes.sql
CREATE INDEX
```

298.1. Schema Result

We can log back into the database to take a look at the resulting schema. The following executes the

`psql` CLI interface in the postgres image.

Interactive Login to postgres

```
$ docker-compose exec postgres psql -U postgres
psql (12.3)
Type "help" for help.
#
```

298.2. List Tables

The following lists the tables created in the postgres database.

List Tables

```
postgres=# \d+
              List of relations
 Schema |           Name           |   Type   | Owner |     Size    | Description
-----+---------------------+-----+-----+-----+-----+
 public | reposongs_song        | table  | postgres | 16 kB     | song database
 public | reposongs_song_sequence | sequence | postgres | 8192 bytes |
(2 rows)
```

298.3. Describe Song Table

Describe Song Table

```
postgres=# \d reposongs_song
          Table "public.reposongs_song"
 Column |           Type           | Collation | Nullable | Default
-----+-----+-----+-----+-----+
 id    | integer            |           | not null |
 title | character varying(255) |           |           |
 artist | character varying(255) |           |           |
 released | date             |           |           |
Indexes:
 "song_pk" PRIMARY KEY, btree (id)
 "idx_song_title" btree (title)
```

Chapter 299. RDBMS Project

Although it is common to execute schema commands interactively during initial development, sooner or later they should end up documented in source file(s) that can help document the baseline schema and automate getting to a baseline schema state. Spring Boot provides direct support for automating schema migration—whether it be for test environments or actual production migration. This automation is critical to modern dynamic deployment environments. Lets begin filling in some project-level details of our example.

299.1. RDBMS Project Dependencies

To get our project prepared to communicate with the database, we are going to need a RDBMS-based spring-data starter and at least one database dependency.

The following dependency example readies our project for JPA (a layer well above RDBMS) and to be able to use either the `postgres` or `h2` database.

- `h2` is an easy and efficient in-memory database choice to base unit testing. Other in-memory choices include `HSQLDB` and `Derby` databases.
- `postgres` is one of [many choices](#) we could use for a production-ready database

RDBMS Project Dependencies

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId> ①
</dependency>
②
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
<!-- schema management --> ③
<dependency>
    <groupId>org.flywaydb</groupId>
    <artifactId>flyway-core</artifactId>
    <scope>runtime</scope>
</dependency>
```

① brings in all dependencies required to access database using JPA (including APIs and Hibernate implementation)

② defines two database clients we have the option of using—`h2` offers an in-memory server

③ brings in a schema management tool

299.2. RDBMS Access Objects

The JPA starter takes care of declaring a few key `@Bean` instances that can be injected into components.

- `javax.sql.DataSource` is part of the standard JDBC API—which is a very mature and well-supported standard
- `jakarta.persistence.EntityManager` is part of the standard JPA API—which is a layer above JDBC and also a well-supported standard.

Key RDBMS Objects

```
@Autowired  
private javax.sql.DataSource dataSource; ①  
  
@Autowired  
private jakarta.persistence.EntityManager entityManager; ②
```

① `DataSource` defines a starting point to interface to database using JDBC

② `EntityManager` defines a starting point for JPA interaction with the database

299.3. RDBMS Connection Properties

Spring Boot will make some choices automatically, but since we have defined two database dependencies, we should be explicit. The default datasource is defined with the `spring.datasource` prefix. The URL defines which client to use. The driver-class-name and dialect can be explicitly defined, but can also be determined internally based on the URL and details reported by the live database.

The following example properties define an in-memory h2 database.

h2 in-memory database properties

```
spring.datasource.url=jdbc:h2:mem:songs  
#spring.datasource.driver-class-name=org.h2.Driver ①
```

① Spring Boot can automatically determine driver-class-name from provided URL

The following example properties define a postgres client. Since this is a server, we have other properties—like username and password—that have to be supplied.

postgres server database client properties

```
spring.datasource.url=jdbc:postgresql://localhost:5432/postgres  
spring.datasource.username=postgres  
spring.datasource.password=secret  
#spring.datasource.driver-class-name=org.postgresql.Driver
```

```
#spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
```

Driver can be derived from JDBC URL



In a normal Java application, JDBC drivers automatically register with the JDBC DriverManager at startup. When a client requests a connection to a specific JDBC URL, the JDBC DriverManager interrogates each driver, looking for support for the provided JDBC URL.

Chapter 300. Schema Migration

The schema of a project rarely stays constant and commonly has to migrate from version to version. No matter what can be automated during development, we need to preserve existing data in production and formal integration environments. Spring Boot has a default integration with Flyway in order to provide ordered migration from version to version. Some of its features (e.g., undo) require a commercial license, but its open-source offering implements forward migrations for free.

300.1. Flyway Automated Schema Migration

"Flyway is an open-source database migration tool".^[1] It comes [pre-integrated with Spring Boot](#) once we add the Maven module dependency. Flyway executes provided SQL migration scripts against the database and maintains the state of the migration for future sessions.

300.2. Flyway Schema Source

By default, schema files^[2]

- are searched for in the `classpath:db/migration` directory
 - overridden using `spring.flyway.locations` property
 - locations can be from the classpath and filesystem
 - location expressions support `{vendor}` placeholder expansion

```
spring.flyway.locations=classpath:db/migration/common,classpath:db/migration/{vendor}
```

- following a naming pattern of `V<version>_<name/comment>.sql` (double underscore between version and name/comment) with version being a period (".") or single underscore ("_") separated set of version digits (e.g., `V1.0.0_0`, `V1_0_0_0`)

The following example shows a set of schema migration files located in the default, vendor neutral location.

Schema Migration Target Folder

```
target/classes/
|-- application-postgres.properties
|-- application.properties
\-- db
    '-- migration
        |-- V1.0.0_0__initial_schema.sql
        |-- V1.0.0_1__initial_indexes.sql
        '-- V1.1.0_0__add_artist.sql
```

300.3. Flyway Automatic Schema Population

Spring Boot will automatically trigger a migration of the files when the application starts.

The following example is launching the application and activating the `postgres` profile with the client setup to communicate with the remote postgres database. The `--db.populate` is turning off application level population of the database. That is part of a later example.

Active Database Server Profile

```
java -jar target/jpa-song-example-6.1.0-SNAPSHOT-bootexec.jar  
--spring.profiles.active=postgres --db.populate=false
```

300.4. Database Server Profiles

By default, the example application will use an in-memory database.

application.properties

```
#h2  
spring.datasource.url=jdbc:h2:mem:users
```

To use the postgres database, we need to fill in the properties within the selected profile.

application-postgres.properties

```
#postgres  
spring.datasource.url=jdbc:postgresql://localhost:5432/postgres  
spring.datasource.username=postgres  
spring.datasource.password=secret
```

300.5. Dirty Database Detection

If flyway detects a non-empty schema and no flyway table(s), it will immediately throw an exception and the program terminates.

Flyway Detects an Error

```
FlywayException: Found non-empty schema(s) "public" but no schema history table.  
Use baseline() or set baselineOnMigrate to true to initialize the schema history  
table.
```

Keeping this simple, we can simply drop the existing schema.

Drop Existing

```
postgres=# drop table reposongs_song;
```

```
DROP TABLE
postgres=# drop sequence reposongs_song_sequence;
DROP SEQUENCE
```

300.6. Flyway Migration

With everything correctly in place, flyway will execute the migration.

The following output is from the console log showing the activity of Flyway migrating the schema of the database.

Flyway Migration Debug Log Statements

```
VersionPrinter : Flyway Community Edition 7.1.1 by Redgate
DatabaseType   : Database: jdbc:postgresql://localhost:5432/postgres (PostgreSQL 12.3)
DbValidate     : Successfully validated 3 migrations (execution time 00:00.026s)
JdbcTableSchemaHistory : Creating Schema History table
"public"."flyway_schema_history" ...
DbMigrate : Current version of schema "public": << Empty Schema >>
DbMigrate : Migrating schema "public" to version "1.0.0.0 - initial schema"
DefaultSqlScriptExecutor : DB: sequence "reposongs_song_sequence" does not exist,
skipping
DefaultSqlScriptExecutor : DB: table "reposongs_song" does not exist, skipping
DbMigrate : Migrating schema "public" to version "1.0.0.1 - initial indexes"
DbMigrate : Migrating schema "public" to version "1.1.0.0 - add artist"
DbMigrate : Successfully applied 3 migrations to schema "public" (execution time
00:00.190s)
```

[1] "Flyway Documentation",[Flyway Web Page](#)

[2] "Execute Flyway Database Migrations on Startup",[docs.spring.io Web Site](#)

Chapter 301. SQL CRUD Commands

All RDBMS-based interactions are based on Structured Query Language (SQL) and its set of Data Manipulation Language (DML) commands. It will help our understanding of what the higher-level frameworks are providing if we take a look at a few raw examples.



SQL Commands are case-insensitive

All SQL commands are case-insensitive. Using upper or lower case in these examples is a matter of personal/project choice.

301.1. H2 Console Access

When H2 is activated — we can activate the H2 user interface using the following property.

Activating H2 Console

```
spring.h2.console.enabled=true
```

Once the application is up and running, the following URL provides access to the H2 console.

Accessing H2 Console

```
http://localhost:8080/h2-console
```

Table 22. H2 Console Windows

The screenshot shows two windows side-by-side. On the left is the 'Login' window, which has fields for 'Saved Settings' (set to 'Generic H2 (Embedded)'), 'Setting Name' (set to 'Generic H2 (Embedded)'), 'Driver Class' (set to 'org.h2.Driver'), 'JDBC URL' (set to 'jdbc:h2:mem:users'), 'User Name' (set to 'sa'), and 'Password'. There are 'Connect' and 'Test Connection' buttons at the bottom. On the right is the 'REPOSONGS_SONG' schema browser window. It shows tables like 'ID', 'TITLE', 'RELEASED', 'ARTIST', and 'Indexes'. It also shows sequences like 'hibernate_sequence' and 'NEXT VALUE FOR PUBLIC.HIBERNATE_SEQUENCE' which is set to '1'. The top of the browser window has buttons for 'Run', 'Run Selected', 'Auto complete', 'Clear', and 'SQL statement'.

301.2. Postgres CLI Access

With postgres activated, we can access the postgres server using the `psql` CLI.

Postgres Command Line Interface Access

```
$ docker-compose exec postgres psql -U postgres
psql (12.3)
Type "help" for help.
```

```
postgres=#
```

301.3. Next Value for Sequence

We created a sequence in our schema to manage unique IDs. We can obtain the next value for that sequence using a SQL command. Unfortunately, obtaining the next value for a sequence is vendor-specific. The following two examples show examples for postgres and h2.

postgres sequence next value example

```
select nextval('reposongs_song_sequence');
nextval
-----
6
```

h2 sequence next value example

```
call next value for reposongs_song_sequence;
---
1
```

301.4. SQL ROW INSERT

We add data to a table using the INSERT command.

SQL INSERT Example

```
insert into reposongs_song(id, title, artist, released)
values (6,'Don''t Worry Be Happy','Bobby McFerrin', '1988-08-05');
```

Use two single-quote characters to embed single-quote



The single-quote character is used to delineate a string in SQL commands. Use two single-quote characters to express a single quote character within a command (e.g., `Don''t`).

301.5. SQL SELECT

We output row data from the table using the SELECT command;

SQL SELECT Wildcard Example

```
# select * from reposongs_song;

id |      title       |     artist    | released
---+-----+-----+-----+
 6 | Don't Worry Be Happy | Bobby McFerrin | 1988-08-05
```

The previous example output all columns and rows for the table in a non-deterministic order. We can control the columns output, the column order, and the row order for better management. The next example outputs specific columns and orders rows in ascending order by the released date.

SQL SELECT Columns and Order Example

```
# select released, title, artist from reposongs_song order by released ASC;
released | title | artist
-----+-----+-----
1986-05-18 | Sledgehammer | Peter Gabriel
1988-08-05 | Don't Worry Be Happy | Bobby McFerrin
```

301.6. SQL ROW UPDATE

We can change column data of one or more rows using the UPDATE command.

The following example shows a row with a value that needs to be changed.

Incorrect Row

```
# insert into reposongs_song(id, title, artist, released)
values (8,'October','Earth Wind and Fire', '1978-11-18');
```

The following snippet shows updating the title column for the specific row.

SQL UPDATE Example

```
# update reposongs_song set title='September' where id=8;
```

The following snippet uses the SELECT command to show the results of our change.

SQL UPDATE Result

```
# select * from reposongs_song where id=8;
id | title | artist | released
---+-----+-----+-----
8 | September | Earth Wind and Fire | 1978-11-18
```

301.7. SQL ROW DELETE

We can remove one or more rows with the DELETE command. The following example removes a specific row matching the provided ID.

SQL DELETE Example

```
# delete from reposongs_song where id=8;  
DELETE 1
```

```
# select * from reposongs_song;  
id | title | artist | released  
---+-----+-----+-----  
6 | Don't Worry Be Happy | Bobby McFerrin | 1988-08-05  
7 | Sledgehammer | Peter Gabriel | 1986-05-18
```

301.8. RDBMS Transaction

Transactions are an important and integral part of relational databases. The transactionality of a database are expressed in "ACID" properties [\[1\]](#):

- Atomic - all or nothing. Everything in the unit acts as a single unit
- Consistent - moves from one valid state to another
- Isolation - the degree of visibility/independence between concurrent transactions
- Durability - a committed transaction exists

By default, most interactions with the database are considered individual transactions with an auto-commit after each one. Auto-commit can be disabled so that multiple commands can be part of the same, single transaction.

301.8.1. BEGIN Transaction Example

The following shows an example of a disabling auto-commit in postgres by issuing the BEGIN command. Every change from this point until the COMMIT or ROLLBACK is temporary and is isolated from other concurrent transactions (to the level of isolation supported by the database and configured by the connection).

BEGIN Transaction Example

```
# BEGIN; ①  
BEGIN  
  
# insert into reposongs_song(id, title, artist, released)  
values (7,'Sledgehammer','Peter Gabriel', '1986-05-18');  
INSERT 0 1  
  
# select * from reposongs_song;  
id | title | artist | released | foo  
---+-----+-----+-----+  
6 | Don't Worry Be Happy | Bobby McFerrin | 1988-08-05 |  
7 | Sledgehammer | Peter Gabriel | 1986-05-18 | ②
```

(3 rows)

- ① new transaction started when BEGIN command issued
- ② commands within a transaction will be able to see uncommitted changes from the same transaction

301.8.2. ROLLBACK Transaction Example

The following shows how the previous command(s) in the current transaction can be rolled back—as if they never executed. The transaction ends once we issue COMMIT or ROLLBACK.

ROLLBACK Example

```
# ROLLBACK; ①
ROLLBACK

# select * from reposongs_song; ②
id | title | artist | released
---+-----+-----+-----
 6 | Don't Worry Be Happy | Bobby McFerrin | 1988-08-05
```

- ① transaction ends once COMMIT or ROLLBACK command issued
- ② commands outside of a transaction will not be able to see uncommitted and rolled back changes of another transaction

Chapter 302. JDBC

With database schema in place and a key amount of SQL under our belt, it is time to move on to programmatically interacting with the database. Our next stop is a foundational aspect of any Java database interaction, the Java Database Connectivity (JDBC) API. JDBC is a standard Java API for communicating with tabular databases.^[1] We hopefully will never need to write this code in our applications, but it eventually gets called by any database mapping layers we may use—therefore it is good to know some of the foundation.

302.1. JDBC DataSource

The `javax.sql.DataSource` is the starting point for interacting with the database. Assuming we have Flyway schema migrations working at startup, we already know we have our database properties setup properly. It is now our chance to inject a `DataSource` and do some work.

The following snippet shows an example of an injected `DataSource`. That `DataSource` is being used to obtain the URL used to connect to the database. Most JDBC commands declare a checked exception (`SQLException`) that must be caught or also declared thrown.

Injecting a DataSource

```
@Component  
@RequiredArgsConstructor  
public class JdbcSongDAO {  
    private final javax.sql.DataSource dataSource; ①  
  
    @PostConstruct  
    public void init() {  
        try (Connection conn=dataSource.getConnection()) {  
            String url = conn.getMetaData().getURL();  
            ... ②  
        } catch (SQLException ex) { ③  
            throw new IllegalStateException(ex);  
        }  
    }  
}
```

① `DataSource` injected using constructor injection

② `DataSource` used to obtain a connection and metadata for the URL

③ Most JDBC commands declare throwing an `SQLException` that must be explicitly handled

302.2. Obtain Connection and Statement

We obtain a `java.sql.Connection` from the `DataSource` and a `Statement` from the connection. Connections and statements must be closed when complete and we can automate that with a `Java try-with-resources` statement. `PreparedStatement` can be used to assemble the statement up front and reused in a loop if appropriate.

```

public void create(Song song) throws SQLException {
    String sql = //insert/select/delete/update ... ①

    try(Connection conn = dataSource.getConnection(); ②
        PreparedStatement statement = conn.prepareStatement(sql)) { ②

        //statement.executeUpdate(); ③
        //statement.executeQuery();
    }
}

```

① action-specific SQL will be supplied to the `PreparedStatement`

② try-with-resources construct automatically closes objects declared at this scope

③ `Statement` used to query and modify database

The try-with-resources construct auto-closes the resources in reverse order of declaration.

Try-with-resources Can Accept Instances

The try-with-resources can be refactored to take just the `Autocloseable` variable if preferred.



```

Connection conn = dataSource.getConnection();
//logic
PreparedStatement statement = conn.prepareStatement(sql);
//logic
try(conn; statement) { ①
    //statement.method();
}

```

① try-with-resources will close these instances when complete

302.3. JDBC Create Example

JDBC Create Example

```

public void create(Song song) throws SQLException {
    String sql = "insert into REPOSONGS_SONG(id, title, artist, released)
values(?, ?, ?, ?); ①

    try(Connection conn = dataSource.getConnection();
        PreparedStatement statement = conn.prepareStatement(sql)) {
        int id = nextId(conn); //get next ID from database ②
        log.info("{} , params={{}}, sql, List.of(id, song.getTitle(), song.getArtist(),
song.getReleased()));

        statement.setInt(1, id); ③
        statement.setString(2, song.getTitle());
    }
}

```

```

        statement.setString(3, song.getArtist());
        statement.setDate(4, Date.valueOf(song.getReleased()));
        statement.executeUpdate();

        setId(song, id); //inject ID into supplied instance ④
    }
}

```

- ① SQL commands have ? placeholders for parameters
- ② leveraging a helper method (based on a query statement) to obtain next sequence value
- ③ filling in the individual variables of the SQL template
- ④ leveraging a helper method (based on Java reflection) to set the generated ID of the instance before returning

Use Variables over String Literal Values



Repeated SQL commands should always use parameters over literal values. Identical SQL templates allow database parsers to recognize a repeated command and leverage earlier query plans. Unique SQL strings require database to always parse the command and come up with new query plans.

302.4. Set ID Example

The following snippet shows the helper method used earlier to set the ID of an existing instance. We need the helper because `id` is declared private. `id` is declared private and without a setter because it should never change. Persistence is one of the exceptions to "should never change".

Example Helper Method to Set Private ID of instance

```

private void setId(Song song, int id) {
    try {
        Field f = Song.class.getDeclaredField("id"); ①
        f.setAccessible(true); ②
        f.set(song, id); ③
    } catch (NoSuchFieldException | IllegalAccessException ex) {
        throw new IllegalStateException("unable to set Song.id", ex);
    }
}

```

- ① using Java reflection to locate the `id` field of the `Song` class
- ② must set to accessible since `id` is private — otherwise an `IllegalAccessException`
- ③ setting the value of the `id` field

302.5. JDBC Select Example

The following snippet shows an example of using a JDBC select. In this case we are querying the database and representing the returned rows as instances of `Song` POJOs.

JDBC Select Example

```
public Song findById(int id) throws SQLException {
    String sql = "select title, artist, released from REPOSONGS_SONG where id=?"; ①

    try(Connection conn = dataSource.getConnection();
        PreparedStatement statement = conn.prepareStatement(sql)) {
        statement.setInt(1, id); ②
        try (ResultSet rs = statement.executeQuery()) { ③
            if (rs.next()) { ④
                Date releaseDate = rs.getDate(3); ⑤
                return Song.builder()
                    .id(id)
                    .title(rs.getString(1))
                    .artist(rs.getString(2))
                    .released(releaseDate == null ? null : releaseDate.toLocalDate())
                    .build();
            } else {
                throw new NoSuchElementException(String.format("song[%d] not found",
id));
            }
        }
    }
}
```

① provide a SQL template with ? placeholders for runtime variables

② fill in variable placeholders

③ execute query and process results in one or more `ResultSet`—which must be closed when complete

④ must test `ResultSet` before obtaining first and each subsequent row

⑤ obtain values from the `ResultSet`—numerical order is based on SELECT clause

302.6. nextId

The `nextId()` call from `createSong()` is another query on the surface, but it is incrementing a sequence at the database level to supply the value.

`nextId`

```
private int nextId(Connection conn) throws SQLException {
    String sql = dialect.getNextvalSql();
    log.info(sql);
    try(PreparedStatement call = conn.prepareStatement(sql);
        ResultSet rs = call.executeQuery()) {
        if (rs.next()) {
            Long id = rs.getLong(1);
            return id.intValue();
        } else {
```

```
        throw new IllegalStateException("no sequence result returned from call");
    }
}
}
```

302.7. Dialect

Sequences syntax (and support for Sequences) is often DB-specific. Therefore, if we are working at the SQL or JDBC level, we need to use the proper dialect for our target database. The following snippet shows two choices for dialect for getting the next value for a sequence.

Dialect

```
private Dialect dialect;

enum Dialect {
    H2("call next value for reposongs_song_sequence"),
    POSTGRES("select nextval('reposongs_song_sequence')");

    private String nextvalSql;
    private Dialect(String nextvalSql) {
        this.nextvalSql = nextvalSql;
    }
    String getNextvalSql() { return nextvalSql; }
}
```

Chapter 303. Summary

In this module, we learned:

- to define a relational database schema for a table, columns, sequence, and index
- to define a primary key, table constraints, and an index
- to automate the creation and migration of the database schema
- to interact with database tables and columns with SQL
- underlying JDBC API interactions

Java Persistence API (JPA)

copyright © 2024 jim stafford (jim.stafford@jhu.edu)

Chapter 304. Introduction

This lecture covers implementing object/relational mapping (ORM) to an RDBMS using the Java Persistence API (JPA). This lecture will directly build on the previous concepts covered in the RDBMS and show the productivity power gained by using an ORM to map Java classes to the database.

304.1. Goals

The student will learn:

- to identify the underlying JPA constructs that are the basis of Spring Data JPA Repositories
- to implement a JPA application with basic CRUD capabilities
- to understand the significance of transactions when interacting with JPA

304.2. Objectives

At the conclusion of this lecture and related exercises, the student will be able to:

1. declare project dependencies required for using JPA
2. define a `DataSource` to interface with the RDBMS
3. define a Persistence Context containing an `@Entity` class
4. inject an `EntityManager` to perform actions on a PersistenceUnit and database
5. map a simple `@Entity` class to the database using JPA mapping annotations
6. perform basic database CRUD operations on an `@Entity`
7. define transaction scopes

Chapter 305. Java Persistence API

The Java Persistence API (JPA) is an object/relational mapping (ORM) layer that sits between the application code and JDBC and is the basis for Spring Data JPA Repositories. JPA permits the application to primarily interact with plain old Java (POJO) business objects and a few standard persistence interfaces from JPA to fully manage our objects in the database. JPA works off convention and customized by annotations primarily on the POJO, called an Entity. JPA offers a rich set of capabilities that would take us many chapters and weeks to cover. I will just cover the very basic setup and `@Entity` mapping at this point.

305.1. JPA Standard and Providers

The JPA standard was originally part of Java EE, which is now managed by the [Eclipse Foundation within Jakarta](#). It was released just after Java 5, which was the first version of Java to support annotations. It replaced the older, heavyweight Entity Bean Standard—that was ill-suited for the job of realistic O/R mapping—and progressed on a path in line with Hibernate. There are several [persistence providers of the API](#)

- [EclipseLink](#) is now the reference implementation
- [Hibernate](#) was one of the original implementations and the default implementation within Spring Boot
- [DataNucleus](#)

305.2. Javax / Jakarta Transition

JPA transitioned from Oracle/JavaEE to Jakarta in ~2019 and was officially renamed from "Java Persistence API" to "Jakarta Persistence". Official references dropped the "API" portion of the name and used the long name. Unofficial references keep the "API" portion of the name and still reference the product as "JPA". For simplicity—I will take the unofficial reference route and continue to refer to the product as "JPA" for short once we finish this background paragraph.

The [last release](#) of the "Java Persistence API" (officially known as "JPA") was 2.2 in ~2017. You will never see a version of the "Java Persistence API" newer than 2.2.x. "Jakarta Persistence" (unofficially known as "JPA") [released a clone](#) of the "Java Persistence API" version 2.2.x under the [jakarta](#) Maven package naming to help in the transition. "Jakarta Persistence" 3.0 was released in ~2020 with just package renaming from `javax.persistence` to `jakarta.persistence`. Enhancements were not added until JPA 3.1 in ~2022. That means the feature set remained idle for close to 5 years during the transition.

JPA Maven Module and Java Packaging Transition

- [javax.persistence:javax.persistence-api](#)
 - used javax.persistence Java package naming
 - ended with version 2.2 in 2017
- [jakarta.persistence:jakarta.persistence-api](#) 2.2.x



- used javax.persistence Java package naming
- released in 2018
- jakarta.persistence:jakarta.persistence-api 3.x
 - uses jakarta.persistence Java package naming
 - released in 2020

Spring Boot 2.7 advanced to `jakarta.persistence:jakarta.persistence-api:jar:2.2.3`, which kept the `javax.persistence` Java package naming. Spring Boot 3 advanced to `jakarta.persistence:jakarta.persistence-api:jar:3.x.x`, which required all Java package references to change from `javax.persistence` to `jakarta.persistence`. With a port to Spring Boot 3/Spring 6, we have finally turned the corner on the `javax /jakarta` naming issues.

Spring Boot Javax/Jakarta Transition



- Spring Boot 2.7
 - uses jakarta.persistence:jakarta.persistence-api 2.2.x
 - with javax.persistence Java package naming
- Spring Boot 3
 - uses jakarta.persistence:jakarta.persistence-api 3.x
 - with jakarta.persistence Java package naming

305.3. JPA Dependencies

Access to JPA requires declaring a dependency on the JPA interface (`jakarta.persistence-api`) and a provider implementation (e.g., `hibernate-core`). This is automatically added to the project by declaring a dependency on the `spring-boot-starter-data-jpa` module.

Spring Data JPA Maven Dependency

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

The following shows a subset of the dependencies brought into the application by declaring a dependency on the JPA starter.

Spring Boot Starter JPA 3.1 Dependencies

```
+-- org.springframework.boot:spring-boot-starter-data-jpa:jar:3.3.2:compile
|   +- org.springframework.boot:spring-boot-starter-jdbc:jar:3.3.2:compile
...
|   |   \- org.springframework:spring-jdbc:jar:6.1.11:compile
|   +- org.hibernate.orm:hibernate-core:jar:6.5.2.Final:compile ②
|   |   +- jakarta.persistence:jakarta.persistence-api:jar:3.1.0:compile ①
```

```
| | +- jakarta.transaction:jakarta.transaction-api:jar:2.0.1:compile
```

```
...
```

- ① the JPA API module is required to compile standard JPA constructs
- ② a JPA provider module is required to access extensions and for runtime implementation of the standard JPA constructs

Spring Boot Starter JPA 2.7 Dependencies

```
+- org.springframework.boot:spring-boot-starter-data-jpa:jar:2.7.0:compile
| +- org.springframework.boot:spring-boot-starter-jdbc:jar:2.7.0:compile
|   | \- org.springframework:spring-jdbc:jar:5.3.20:compile
| +- jakarta.persistence:jakarta.persistence-api:jar:2.2.3:compile ①
| +- org.hibernate:hibernate-core:jar:5.6.9.Final:compile ②
...

```

- ① the JPA API module is required to compile standard JPA constructs
- ② a JPA provider module is required to access extensions and for runtime implementation of the standard JPA constructs

From these dependencies, we can define and inject various JPA beans.

305.4. Enabling JPA AutoConfiguration

JPA has its own defined bootstrapping constructs that involve settings in `persistence.xml` and entity mappings in `orm.xml` configuration files. These files define the overall persistence unit and include information to connect to the database and any custom entity mapping overrides.

Spring Boot JPA automatically configures a default persistence unit and other related beans when the `@EnableJpaRepositories` annotation is provided. `@EntityScan` is used to identify packages for `@Entities` to include in the persistence unit.

Spring Boot Data Bootstrapping

```
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;

@SpringBootApplication
@EnableJpaRepositories ①
// Class<?>[] basePackageClasses() default {};
// String repositoryImplementationPostfix() default "Impl";
// ... (many more configurations)
@EntityScan ②
// Class<?>[] basePackageClasses() default {};
public class JPASongsApp {
```

- ① triggers and configures scanning for JPA Repositories (the topic of the next lecture)
- ② triggers and configures scanning for JPA Entities

By default, this configuration will scan packages below the class annotated with the `@EntityScan` annotation. We can override that default using the attributes of the `@EntityScan` annotation.

305.5. Configuring JPA DataSource

Spring Boot provides convenient ways to provide property-based configurations through its standard property handing, making the connection areas of `persistence.xml` unnecessary (but still usable). The following examples show how our definition of the `DataSource` for the JDBC/SQL example can be used for JPA as well.

Table 23. Spring Data JPA Database Connection Properties

H2 In-Memory Example Properties

```
spring.datasource.url=jdbc:h2:mem:songs
```

Postgres Client Example Properties

```
spring.datasource.url=jdbc:postgresql://localhost:5432/postgres
spring.datasource.username=postgres
spring.datasource.password=secret
```

305.6. Automatic Schema Generation

JPA provides the capability to automatically generate schema from the Persistence Unit definitions. This can be configured to write to a file to be used to kickstart schema authoring. However, the most convenient use for schema generation is at runtime during development.

Spring Boot will automatically enable runtime schema generation for in-memory database URLs. We can also explicitly enable runtime schema generation using the following Hibernate property.

Example Explicit Enable Runtime Schema Generation

```
spring.jpa.hibernate.ddl-auto=create
```

305.7. Schema Generation to File

The JPA provider can be configured to generate schema to a file. This can be used directly by tools like Flyway or simply to kickstart manual schema authoring.

The following configuration snippet instructs the JPA provider to generate a create and drop commands into the same `drop_create.sql` file based on the metadata discovered within the Persistence Context. Hibernate has the additional features to allow for formatting and line termination specification.

Schema Generation to File Example

```
spring.jpa.properties.jakarta.persistence.schema-generation.scripts.action=drop-and-
```

```

create
spring.jpa.properties.jakarta.persistence.schema-generation.create-source=metadata

spring.jpa.properties.jakarta.persistence.schema-generation.scripts.create-
target=target/generated-sources/ddl/drop_create.sql
spring.jpa.properties.jakarta.persistence.schema-generation.scripts.drop-
target=target/generated-sources/ddl/drop_create.sql

spring.jpa.properties.hibernate.hbm2ddl.delimiter=; ①
spring.jpa.properties.hibernate.format_sql=true ②

```

① adds ";" character to terminate every command — making it SQL script-ready

② adds new lines to make more human-readable

`action` can have values of `none`, `create`, `drop-and-create`, and `drop` ^[1]

`create/drop-source` can have values of `metadata`, `script`, `metadata-then-script`, or `script-then-metadata`. `metadata` will come from the class defaults and annotations. `script` will come from a location referenced by `create/drop-script-source`



Generate Schema to Debug Complex Mappings

Generating schema from `@Entity` class metadata is a good way to debug odd persistence behavior. Even if normally ignored, the generated schema can identify incorrect and accidental definitions that may cause unwanted behavior. I highly recommend activating it while working with new `@Entity` mappings or debugging old ones.

305.8. Generated Schema Output

The following snippet shows the output of the Entity Manager when provided with the property activations and configuration. The specific output contains all SQL commands required to drop existing rows/schema (by the same name(s)) and instantiate a new version based upon the `@Entity` metadata.

Generated Schema Output

```

$ docker compose up postgres -d ①
$ mvn clean package spring-boot:start spring-boot:stop \
-Dspring-boot.run.profiles=postgres -DskipTests ②

$ cat target/generated-sources/ddl/drop_create.sql ③

    drop table if exists reposongs_song cascade;

    drop sequence if exists reposongs_song_sequence;

    create sequence reposongs_song_sequence start with 1 increment by 50;

    create table reposongs_song (

```

```

    id integer not null,
    released date,
    artist varchar(255),
    title varchar(255),
    primary key (id)
);

```

- ① starting Postgres DB
- ② starting/stopping server with JPA EntityManager
- ③ JPA EntityManager-produced schema file

305.9. Other Useful Properties

It is useful to see database SQL commands coming from the JPA/Hibernate layer during early stages of development or learning. The following properties will print the JPA SQL commands and values mapped to the SQL substitution variables.

The first two property settings both functionally produce logging of SQL statements.

Two JPA/Hibernate SQL Logging Options

```

spring.jpa.show-sql=true
logging.level.org.hibernate.SQL=DEBUG

```

- The `spring.jpa.show-sql` property controls raw text output, lacking any extra clutter. This is a good choice if you want to see the SQL command and do not need a timestamp or threadId.

`spring.jpa.show-sql=true`

```

Hibernate: delete from REPOSONGS_SONG
Hibernate: select next value for reposongs_song_sequence
Hibernate: insert into reposongs_song (artist,released,title,id) values (?, ?, ?, ?)
Hibernate: select count(s1_0.id) from reposongs_song s1_0 where s1_0.id=?

```

- The `logging.level.org.hibernate.SQL` property controls SQL logging passed through the logging framework. This is a good choice if you are looking at a busy log with many concurrent threads. The threadId provides you with the ability to filter the associated SQL commands. The timestamps provide a basic ability to judge performance timing.

`logging.level.org.hibernate.SQL=debug`

```

09:53:38.632 main DEBUG org.hibernate.SQL#logStatement:135 delete from
REPOSONGS_SONG
09:53:38.720 main DEBUG org.hibernate.SQL#logStatement:135 select next value for
reposongs_song_sequence
09:53:38.742 main DEBUG org.hibernate.SQL#logStatement:135 insert into
reposongs_song (artist,released,title,id) values (?, ?, ?, ?)
09:53:38.790 main DEBUG org.hibernate.SQL#logStatement:135 select count(s1_0.id)

```

```
from reposongs_song s1_0 where s1_0.id=?
```

An additional property can be defined to show the mapping of data to/from the database rows. The `logging.level.org.hibernate.orm.jdbc.bind` property controls whether Hibernate prints the value of binding parameters (e.g., "?") input to or returning from SQL commands. It is quite helpful if you are not getting the field data you expect.

JPA/Hibernate JDBC Parameter Mapping Debug Property

```
logging.level.org.hibernate.orm.jdbc.bind=TRACE
```

The following snippet shows the result information of the activated debug. We can see the individual SQL commands issued to the database as well as the parameter values used in the call and extracted from the response. The extra logging properties have been redacted from the output.

```
logging.level.org.hibernate.orm.jdbc.bind=trace
```

```
insert into reposongs_song (artist,released,title,id) values (?,?,?,?,?)  
  
binding parameter (1:VARCHAR) <- [The Orb]  
binding parameter (2:DATE) <- [2020-08-01]  
binding parameter (3:VARCHAR) <- [Mother Night teal]  
binding parameter (4:INTEGER) <- [1]
```

Bind Logging Moved Packages

Hibernate moved some classes. `org.hibernate.type` logging changed to `org.hibernate.orm.jdbc.bind` in version 6.



```
logging.level.org.hibernate.type=TRACE          #older  
logging.level.org.hibernate.orm.jdbc.bind=TRACE #newer
```

305.10. Configuring JPA Entity Scan

Spring Boot JPA will automatically scan for `@Entity` classes. We can provide a specification to external packages to scan using the `@EntityScan` annotation.

The following shows an example of using a String package specification to a root package to scan for `@Entity` classes.

`@EntityScan example`

```
import org.springframework.boot.autoconfigure.domain.EntityScan;  
...  
@EntityScan(value={"info.ejava.examples.db.repo.jpa.songs.bo"})
```

The following example, instead uses a Java class to express a package to scan. We are using a

specific `@Entity` class in this case, but some may define an interface simply to help mark the package and use that instead. The advantage of using a Java class/interface is that it will work better when refactoring.

@EntityScan .class Example

```
import info.ejava.examples.db.repo.jpa.songs.bo.Song;  
...  
@EntityScan(basePackageClasses = {Song.class})
```

305.11. JPA Persistence Unit

The JPA Persistence Unit represents the overall definition of a group of Entities and how we interact with the database. A defined Persistence Unit can be injected into the application using an `EntityManagerFactory`. From this injected instance, clients can gain access to metadata and initiate a Persistence Context.

Persistance Unit/EntityManagerFactory Injection Example

```
import jakarta.persistence.EntityManagerFactory;  
...  
@Autowired  
private EntityManagerFactory emf;
```

It is rare that you will ever need an `EntityManagerFactory` and could be a sign that you may not understand yet what it is meant for versus an injected `EntityManager`. `EntityManagerFactory` is used to create a custom persistence context and transaction. The caller is responsible for beginning and committing the transaction—just like with JDBC. This sometimes is useful to create a well-scoped transaction within a transaction. It is a manual process, outside the management of Spring.

Your first choice should be to inject an `EntityManager`.

Primarily use EntityManager versus EntityManagerFactory



It is rare that one requires the use of an `EntityManagerFactory` and using one places many responsibilities on the caller "to do the right thing". This should be used for custom transactions. Normal business transactions should be handled using a JPA Persistence Context, with the injection of an `EntityManager`.

305.12. JPA Persistence Context

A Persistence Context is a usage instance of a Persistence Unit and is represented by an `EntityManager`. An injected `EntityManager` provides a first-layer cache of entities for business logic to share and transaction(s) to commit or rollback their state together. An `@Entity` with the same identity is represented by a single instance within a Persistence Context/`EntityManager`.

Persistance Context/EntityManager Injection Example

```
import jakarta.persistence.EntityManager;  
...  
@Autowired  
private EntityManager em;
```

Injected `EntityManager`s reference the same Persistence Context when called within the same thread. That means that a `Song` loaded by one client with ID=1 will be available to sibling code when using ID=1.

Use/Inject EntityManagers



Normal application code that creates, gets, updates, and deletes `@Entity` data should use an injected `EntityManager` and allow the transaction management to occur at a higher level.

[1] "JavaEE: The JavaEE Tutorial, Database Schema Creation", Oracle, JavaEE 7

Chapter 306. JPA Entity

A JPA `@Entity` is a class mapped to the database that primarily represents a row in a table. The following snippet is the example `Song` class we have already manually mapped to the `REPOSONGS_SONG` database table using manually written schema and JDBC/SQL commands in a previous lecture. To make the class an `@Entity`, we must:

- annotate the class with `@Entity`
- provide a no-argument constructor
- identify one or more columns to represent the primary key using the `@Id` annotation
- override any convention defaults with further annotations

JPA Example Entity

```
@jakarta.persistence.Entity ①
@Getter
@AllArgsConstructor
@NoArgsConstructor ②
public class Song {
    @jakarta.persistence.Id ③ ④
    private int id;
    @Setter
    private String title;
    @Setter
    private String artist;
    @Setter
    private java.time.LocalDate released;
}
```

① class must be annotated with `@Entity`

② class must have a no-argument constructor

③ class must have one or more fields designated as the primary key

④ annotations can be on the field or property and the choice for `@Id` determines the default

Primary Key property is not modifiable

This Java class is not providing a setter for the field mapped to the primary key in the database. The primary key will be generated by the persistence provider at runtime and assigned to the field. The field cannot be modified while the instance is managed by the provider. The all-args constructor can be used to instantiate a new object with a specific primary key.



306.1. JPA `@Entity` Defaults

By convention and supplied annotations, the class as shown above would:

- have the entity name "Song" (important when expressing JPA queries; ex. `select s from Song s`)

- be mapped to the `SONG` table to match the entity name
- have columns `id integer`, `title varchar`, `artist varchar`, and `released (date)`
- use `id` as its primary key and manage that using a provider-default mechanism

306.2. JPA Overrides

All the convention defaults can be customized by further annotations. We commonly need to:

- supply a table name that matches our intended schema (i.e., `select * from REPOSONGS_SONG` vs `select * from SONG`)
- select which primary key mechanism is appropriate for our use. JPA 3 no longer allows an unnamed generator for a `SEQUENCE`.
- define the `SEQUENCE` to be consistent with our SQL definition earlier
- supply column names that match our intended schema
- identify which properties are optional, part of the initial `INSERT`, and `UPDATE`-able
- supply other parameters useful for schema generation (e.g., String length)

Common JPA Annotation Overrides

```

@Entity
@Table(name="REPOSONGS_SONG") ①
@NoArgsConstructor
...
@SequenceGenerator(name="REPOSONGS_SONG_SEQUENCE", allocationSize = 50)③
public class Song {
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE,
                   generator = "REPOSONGS_SONG_SEQUENCE")②
    @Column(name = "ID") ④
    private int id;
    @Column(name="TITLE", length=255, nullable=true, insertable=true, updatable=true
)⑤
    private String title;
    private String artist;
    private LocalDate released;
}

```

① overriding the default table name `SONG` with `REPOSONGS_SONG`

② overriding the default primary key mechanism with `SEQUENCE`. JPA 3 no longer permits an unnamed sequence generator.

③ defining the `SEQUENCE`

④ re-asserting the default convention column name `ID` for the `id` field

⑤ re-asserting many of the default convention column mappings



Schema generation properties aren't used at runtime

Properties like `length` and `nullable` are only used during optional JPA schema generation and are not used at runtime.

Chapter 307. Basic JPA CRUD Commands

JPA provides an API for implementing persistence to the database through manipulation of `@Entity` instances and calls to the `EntityManager`.

307.1. EntityManager persist()

We create a new object in the database by calling `persist()` on the `EntityManager` and passing in an `@Entity` instance that represents something new. This will:

- assign a primary key if configured to do so
- add the instance to the Persistence Context
- make the `@Entity` instance managed from that point forward

The following snippet shows a partial DAO implementation using JPA.

Example EntityManager persist() Call

```
@Component  
@RequiredArgsConstructor  
public class JpaSongDAO {  
    private final EntityManager em; ①  
  
    public void create(Song song) {  
        em.persist(song); ②  
    }  
    ...
```

① using an injected `EntityManager`; it is important that it be injected

② stage for insertion into database; all instance changes managed from this point forward

A database `INSERT` SQL command will be queued to the database as a result of a successful call and the `@Entity` instance will be in a managed state.

Resulting SQL from persist Call()

```
Hibernate: call next value for reposongs_song_sequence  
Hibernate: insert into reposongs_song (artist, released, title, id) values (?, ?, ?, ?)  
binding parameter (1:VARCHAR) <- [The Orb]  
binding parameter (2:DATE) <- [2020-08-01]  
binding parameter (3:VARCHAR) <- [Mother Night teal]  
binding parameter (4:INTEGER) <- [1]
```

In the managed state, any changes to the `@Entity` will result in either the changes be part of:

- a delayed future `INSERT`

- a future **UPDATE** SQL command if changes occur after the **INSERT**

Updates are issued during the next JPA session "flush". JPA session flushes can be triggered manually or automatically prior to or no later than the next commit.

307.2. EntityManager find() By Identity

JPA supplies a means to get the full **@Entity** using its primary key.

Example EntityManager find() Call

```
public Song findById(int id) {
    return em.find(Song.class, id);
}
```

If the instance is not yet loaded into the Persistence Context, **SELECT** SQL command(s) will be issued to the database to obtain the persisted state. The following snippet shows the SQL generated by Hibernate to fetch the state from the database to realize the **@Entity** instance within the JVM.

Resulting SQL from find() Call

```
Hibernate: select
    s1_.id,
    s1_.artist,
    s1_.released,
    s1_.title
  from reposongs_song s1_
 where s1_.id=?
```

From that point forward, the state will be returned from the Persistence Context without the need to get the state from the database.



Find by identity is resolved first through the local cache and second through a database query. Once the **@Entity** is managed, all explicit **find()** by identity calls will be resolved without a database query. That will not be true when using a query (discussed next) with caller-provided criteria.

307.3. EntityManager query

JPA provides many types of queries

- JPA Query Language (officially abbreviated "JPQL"; often called "JPAQL") - a very SQL-like String syntax expressed in terms of **@Entity** classes and relationship constructs
- Criteria Language - a type-safe, Java-centric syntax that avoids String parsing and makes dynamic query building more efficient than query string concatenation and parsing
- Native SQL - the same SQL we would have provided to JDBC

The following snippet shows an example of executing a JPQL Query.

Example EntityManager Query

```
public boolean existsById(int id) {  
    return em.createQuery("select count(s) from Song s where s.id=:id",①  
        Number.class) ②  
        .setParameter("id", id) ③  
        .getSingleResult() ④  
        .longValue()==1L; ⑤  
}
```

- ① JPQL String based on `@Entity` constructs
- ② query call syntax allows us to define the expected return type
- ③ query variables can be set by name or position
- ④ one (mandatory) or many results can be returned from a query
- ⑤ entity exists if row count of rows matching PK is 1. Otherwise, should be 0

The following shows how our JPQL snippet mapped to the raw SQL issued to the database. Notice that our `Song @Entity` reference was mapped to the `REPOSONGS_SONG` database table.

Resulting SQL from Query Call

```
Hibernate: select  
    count(s1_0.id)  
  from reposongs_song s1_0  
 where s1_0.id=?
```

307.4. EntityManager flush()

Not every change to an `@Entity` and call to an `EntityManager` results in an immediate 1:1 call to the database. Some of these calls manipulate an in-memory cache in the JVM and may get issued in a group of other commands at some point in the future. We normally want to allow the `EntityManager` to cache these calls as much as possible. However, there are times (e.g., before making a raw SQL query) where we want to make sure the database has the current state of the cache.

The following snippet shows an example of flushing the contents of the cache after changing the state of a managed `@Entity` instance.

Example EntityManager flush() Call

```
Song s = ... //obtain a reference to a managed instance  
s.setTitle("...");  
em.flush(); //optional!!! will eventually happen at some point
```

Whether it was explicitly issued or triggered internally by the JPA provider, the following snippet shows the resulting `UPDATE` SQL call to change the state of the database to match the Persistence

Context.

Resulting SQL from flush() Call

```
Hibernate: update reposongs_song
    set artist=?, released=?, title=? ①
    where id=?
```

① all fields designated as `updatable=true` are included in the `UPDATE`



flush() does not Commit Changes

Flushing commands to the database only makes the changes available to the current transaction. The result of the commands will not be available to console queries and future transactions until the changes have been committed.

307.5. EntityManager remove()

JPA provides a means to delete an `@Entity` from the database. However, we must have the managed `@Entity` instance loaded in the Persistence Context first to use this capability. The reason for this is that a JPA delete can optionally involve cascading actions to remove other related entities as well.

The following snippet shows how a managed `@Entity` instance can be used to initiate the removal from the database.

Example EntityManager remove() Call

```
public void delete(Song song) {
    em.remove(song);
}
```

The following snippet shows how the remove command was mapped to a SQL `DELETE` command.

Resulting SQL from remove() Call

```
Hibernate: delete from reposongs_song
where id=?
```

307.6. EntityManager clear() and detach()

There are two commands that will remove entities from the Persistence Context. They have their purpose, but know that they are rarely used and can be dangerous to call.

- `clear()` - will remove all entities
- `detach()` - will remove a specific `@Entity`

I only bring these up because you may come across class examples where I am calling `flush()` and `clear()` in the middle of a demonstration. This is purposely mimicking a fresh Persistence Context

within scope of a single transaction.

clear() and detach() Commands

```
em.clear();
em.detach(song);
```

Calling `clear()` or `detach()` will evict all managed entities or targeted managed `@Entity` from the Persistence Context—loosing any in-progress and future modifications. In the case of returning redacted `@Entities`—this may be exactly what you want (you don't want the redactions to remove data from the database).

Use clear() and detach() with Caution



Calling `clear()` or `detach()` will evict all managed entities or targeted managed `@Entity` from the Persistence Context—loosing any in-progress and future modifications.

Chapter 308. Transactions

All commands require some type of transaction when interacting with the database. The transaction can be activated and terminated at varying levels of scope integrating one or more commands into a single transaction.

308.1. Transactions Required for Explicit Changes/Actions

The injected `EntityManager` is the target of our application calls, and the transaction gets associated with that object. The following snippet shows the provider throwing a `TransactionRequiredException` when the calling `persist()` on the injected `EntityManager` when no transaction has been activated.

Example Persist Failure without Transaction

```
@Autowired  
private EntityManager em;  
...  
@Test  
void transaction_missing() {  
    //given - an instance  
    Song song = mapper.map(dtoFactory.make());  
  
    //when - persist is called without a tx, an exception is thrown  
    em.persist(song); ①  
}
```

① `TransactionRequiredException` exception thrown

Exception Thrown when Required Transaction Missing

```
jakarta.persistence.TransactionRequiredException: No EntityManager with actual  
transaction available for current thread - cannot reliably process 'persist' call
```

308.2. Activating Transactions

Although you will find transaction methods on the `EntityManager`, these are only meant for individually managed instances created directly from the `EntityManagerFactory`. Transactions for injected an `EntityManager` are managed by the container and triggered by the presence of a `@Transactional` annotation on a called bean method within the call stack.

This next example annotates the calling `@Test` method with the `@Transactional` annotation to cause a transaction to be active for the three (3) contained `EntityManager` calls.

Example @Transactional Activation

```
import org.springframework.transaction.annotation.Transactional;
```

```

...
@Test
@Transactional ①
void transaction_present_in_caller() {
    //given - an instance
    Song song = mapper.map(dtoFactory.make());

    //when - persist called within caller transaction, no exception thrown
    em.persist(song); ②
    em.flush(); //force DB interaction ②

    //then
    then(em.find(Song.class, song.getId())).isNotNull(); ②
} ③

```

① `@Transactional` triggers an Aspect to activate a transaction for the Persistence Context operating within the current thread

② the same transaction is used on all three (3) `EntityManager` calls

③ the end of the method will trigger the transaction-initiating Aspect to commit (or rollback) the transaction it activated

Nested calls annotated with `@Transactional`, by default, will continue the current transaction.

308.3. Conceptual Transaction Handling

Logically speaking, the transaction handling done on behalf of `@Transactional` is similar to the snippet shown below. However, as complicated as that is—it does not begin to address nested calls. Also note that a thrown `RuntimeException` triggers a rollback and anything else triggers a commit.

Conceptual View of Transaction Handling

```

tx = em.getTransaction();
try {
    tx.begin();
    //call code ②
} catch (RuntimeException ex) {
    tx.setRollbackOnly(); ①
} finally { ②
    if (tx.getRollbackOnly()) {
        tx.rollback();
    } else {
        tx.commit();
    }
}

```

① `RuntimeException`, by default, triggers a rollback

② Normal returns and checked exceptions, by default, trigger a commit

308.4. Activating Transactions in @Components

We can alternatively push the demarcation of the transaction boundary down to the `@Component` methods.

The snippet below shows a DAO `@Component` that designates each of its methods being `@Transactional`. This has the benefit of knowing that each of the calls to `EntityManager` methods will have the required transaction in place. Whether a new one is the right one is a later topic.

@Transactional Component

```
@Component
@RequiredArgsConstructor
@Transactional ①
public class JpaSongDAO {
    private final EntityManager em;

    public void create(Song song) {
        em.persist(song);
    }
    public Song findById(int id) {
        return em.find(Song.class, id);
    }
    public void delete(Song song) {
        em.remove(song);
    }
}
```

① each method will be assigned to a transaction

308.5. Calling @Transactional @Component Methods

The following example shows the calling code invoking methods of the DAO `@Component` in independent transactions. The code works because there really is no dependency between the `INSERT` and `SELECT` to be part of the same transaction, as long as the `INSERT` commits before the `SELECT` transaction starts.

Calling @Component @Transactional Methods

```
@Test
void transaction_present_in_component() {
    //given - an instance
    Song song = mapper.map(dtoFactory.make());

    //when - persist called within component transaction, no exception thrown
    jpaDao.create(song); ①

    //then
    then(jpaDao.findById(song.getId())).isNotNull(); ②
}
```

① `INSERT` is completed in a separate transaction

② `SELECT` completes in follow-on transaction

308.6. @Transactional @Component Methods SQL

The following shows the SQL triggered by the snippet above with the different transactions annotated.

@Transactional Methods Resulting SQL

```
①
Hibernate: insert into reposongs_song (artist,released,title,id) values (?,?,?,?,?)
②
Hibernate: select
    s1_0.id,
    s1_0.artist,
    s1_0.released,
    s1_0.title
  from reposongs_song s1_0
  where s1_0.id=?
```

① transaction 1

② transaction 2

308.7. Unmanaged @Entity

However, we do not always get that lucky—for individual, sequential transactions to play well together. JPA entities follow the notation of managed and unmanaged/detached state.

- Managed entities are actively being tracked by a Persistence Context
- Unmanaged/Detached entities have either never been or no longer associated with a Persistence Context

The following snippet shows an example of where a follow-on method fails because the `EntityManager` requires that `@Entity` be currently managed. However, the end of the `create()` transaction made it detached.

Unmanaged @Entity

```
@Test
void transaction_common_needed() {
    //given a persisted instance
    Song song = mapper.map(dtoFactory.make());
    jpaDao.create(song); //song is detached at this point ①

    //when - removing detached entity we get an exception
    jpaDao.delete(song); ②
```

- ① the first transaction starts and ends at this call
- ② the `EntityManager.remove` operates in a separate transaction with a detached `@Entity` from the previous transaction

The following text shows the error message thrown by the `EntityManager.remove` call when a detached entity is passed in to be deleted.

```
java.lang.IllegalArgumentException: Removing a detached instance
info.ejava.examples.db.repo.jpa.songs.bo.Song#1
```

308.8. Shared Transaction

We can get things to work better if we encapsulate methods behind a `@Service` method defining good transaction boundaries. Lacking a more robust application, the snippet below adds the `@Transactional` to the `@Test` method to have it shared by the three (3) DAO `@Component` calls—making the `@Transactional` annotations on the DAO meaningless.

Shared Transaction

```
@Test
@Transactional ①
void transaction_common_present() {
    //given a persisted instance
    Song song = mapper.map(dtoFactory.make());
    jpaDao.create(song); //song is detached at this point ②

    //when - removing managed entity, it works
    jpaDao.delete(song); ②

    //then
    then(jpaDao.findById(song.getId())).isNull(); ②
}
```

- ① `@Transactional` at the calling method level is shared across all lower-level calls
- ② Each DAO call is executed in the same transaction and the `@Entity` can still be managed across all calls

308.9. `@Transactional` Attributes

There are several attributes that can be set on the `@Transactional` annotation. A few of the more common properties to set include

- propagation - defaults to REQUIRED, proactively activating a transaction if not already present
 - SUPPORTS - lazily initiates a transaction, but fully supported if already active
 - MANDATORY - error if called without an active transaction
 - REQUIRES_NEW - proactively creates a new transaction separate from the caller's

transaction

- NOT_SUPPORTED - nothing within the called method will honor transaction semantics
- NEVER - do not call with an active transaction
- NESTED - may not be supported, but permits nested transactions to complete before returning to calling transaction
- isolation - location to assign JDBC Connection isolation
- readOnly - defaults to false; hints to JPA provider that entities can be immediately detached
- rollback definitions - when to implement non-standard rollback rules

Chapter 309. Summary

In this module, we learned:

- to configure a JPA project, include project dependencies, and required application properties
- to define a Persistence Context and where to scan for `@Entity` classes
- requirements for an `@Entity` class
- default mapping conventions for `@Entity` mappings
- optional mapping annotations for `@Entity` mappings
- to perform basic CRUD operations with the database

Spring Data JPA Repository

copyright © 2024 jim stafford (jim.stafford@jhu.edu)

Chapter 310. Introduction

JDBC/SQL provided a lot of capabilities to interface with the database, but with a significant amount of code required. JPA simplified the mapping, but as you observed with the JPA DAO implementation—there was still a modest amount of boilerplate code. Spring Data JPA Repository leverages the capabilities and power of JPA to map `@Entity` classes to the database but also further eliminates much of the boilerplate code remaining with JPA by leveraging Dynamic Interface Proxy techniques.

310.1. Goals

The student will learn:

- to manage objects in the database using the Spring Data Repository
- to leverage different types of built-in repository features
- to extend the repository with custom features when necessary

310.2. Objectives

At the conclusion of this lecture and related exercises, the student will be able to:

1. declare a `JpaRepository` for an existing JPA `@Entity`
2. perform simple CRUD methods using provided repository methods
3. add paging and sorting to query methods
4. implement queries based on POJO examples and configured matchers
5. implement queries based on predicates derived from repository interface methods
6. implement a custom extension of the repository for complex or compound database access

Chapter 311. Spring Data JPA Repository

Spring Data JPA provides repository support for JPA-based mappings.^[1] We start off by writing no mapping code—just interfaces associated with our `@Entity` and primary key type—and have Spring Data JPA implement the desired code. The Spring Data JPA interfaces are layered—offering useful tools for interacting with the database. Our primary `@Entity` types will have a repository interface declared that inherit from `JpaRepository` and any custom interfaces we optionally define.

 *Figure 135. Spring Data JPA Repository Interfaces*

The extends path was modified some with the latest version of Spring Data Commons, but the `JpaRepository` ends up being mostly the same by the time the interfaces get merged at the bottom of the inheritance tree.

[1] "Spring Data JPA - Reference Documentation"

Chapter 312. Spring Data Repository Interfaces

As we go through these interfaces and methods, please remember that all of the method implementations of these interfaces (except for custom) will be provided for us.

<code>Repository<T, ID></code>	marker interface capturing the <code>@Entity</code> class and primary key type. Everything extends from this type.
<code>CrudRepository<T, ID></code>	depicts many of the CRUD capabilities we demonstrated with the JPA DAO in the previous JPA lecture
<code>PagingAndSortingRepository<T, ID></code>	Spring Data provides some nice end-to-end support for sorting and paging. This interface adds some sorting and paging to the <code>findAll()</code> query method provided in <code>CrudRepository</code> .
<code>ListPagingAndSortingRepository<T, ID></code>	overrides the <code>PagingAndSorting</code> -based <code>Iterable<T></code> return type to be a <code>List<T></code>
<code>ListCrudRepository</code>	overrides all <code>CRUD</code> -based <code>Iterable<T></code> return types with <code>List<T></code>
<code>QueryByExampleExecutor<T></code>	provides query-by-example methods that use prototype <code>@Entity</code> instances and configured matchers to locate matching results
<code>JpaRepository<T, ID></code>	brings together the <code>CrudRepository</code> , <code>PagingAndSortingRepository</code> , and <code>QueryByExampleExecutor</code> interfaces and adds several methods of its own. Unique to JPA, there are methods related to flush and working with JPA references.
<code>SongsRepositoryCustom</code> / <code>SongsRepositoryCustomImpl</code>	we can write our own extensions for complex or compound calls—while taking advantage of an <code>EntityManager</code> and existing repository methods
<code>SongsRepository</code>	our repository inherits from the repository hierarchy and adds additional methods that are automatically implemented by Spring Data JPA

Chapter 313. SongsRepository

All we need to create a functional repository is an `@Entity` class and a primary key type. From our work to date, we know that our `@Entity` is the Song class and the primary key is the primitive `int` type.

313.1. Song @Entity

Song @Entity Example

```
@Entity  
@NoArgsConstructor  
public class Song {  
    @Id //be sure this is jakarta.persistence.Id  
    private int id;
```



Use Correct @Id

There are many `@Id` annotation classes. Be sure to be the correct one for the technology you are currently mapping. In this case, use `jakarta.persistence.Id`.

313.2. SongsRepository

We declare our repository at whatever level of `Repository` is appropriate for our use. It would be common to simply declare it as extending `JpaRepository`.

```
public interface SongsRepository extends JpaRepository<Song, Integer> {}① ②
```

① Song is the repository type

② Integer is used for the primary key type for an `int`



Consider Using Non-Primitive Primary Key Types

Although these lecture notes provide ways to mitigate issues with generated primary keys using a primitive data type, you will find that Spring Data JPA works easier with nullable object types.

Repositories and Dynamic Interface Proxies

Having covered the lectures on Dynamic Interface Proxies and have seen the amount of boilerplate code that exists for persistence—you should be able to imagine how the repositories could be implemented with no up-front, compilation knowledge of the `@Entity` type.

Chapter 314. Configuration



As you may have noticed and will soon see, there is a lot triggered by the addition of the repository interface. You should have the state of your source code in a stable state and committed before adding the repository.

Assuming your repository and entity classes are in a package below the class annotated with `@SpringBootApplication` — all that is necessary is the `@EnableJpaRepositories` to enable the necessary auto-configuration to instantiate the repository.

Typical JPA Repository Support Declaration

```
@SpringBootApplication  
@EnableJpaRepositories  
public class JPASongsApp {
```

If, however, your repository or entities are not located in the default packages scanned, their packages can be scanned with configuration options to the `@EnableJpaRepositories` and `@EntityScan` annotations.

Configuring Repository and @Entity Package Scanning

```
@EnableJpaRepositories(basePackageClasses = {SongsRepository.class}) ① ②  
@EntityScan(basePackageClasses = {Song.class}) ① ③
```

① the Java class provided here is used to identify the base Java package

② where to scan for repository interfaces

③ where to scan for `@Entity` classes

314.1. Injection

With the repository interface declared and the JPA repository support enabled, we can then successfully inject the repository into our application.

SongsRepository Injection

```
@Autowired  
private SongsRepository songsRepo;
```

Chapter 315. CrudRepository

Let's start looking at the capability of our repository—starting with the declared methods of the `CrudRepository` interface and the return type overrides of the `ListCrudRepository` interface.

`CrudRepository<T, ID>` and `ListCrudRepository<T, ID>` Interfaces

```
public interface CrudRepository<T, ID> extends Repository<T, ID> {
    <S extends T> S save(S);
    <S extends T> Iterable<S> saveAll(Iterable<S>);
    Optional<T> findById(ID);
    boolean existsById(ID);
    Iterable<T> findAll();
    Iterable<T> findAllById(Iterable<ID>);
    long count();
    void deleteById(ID);
    void delete(T);
    void deleteAllById(Iterable<? extends ID>);
    void deleteAll(Iterable<? extends T>);
    void deleteAll();
}

public interface ListCrudRepository<T, ID> extends CrudRepository<T, ID> {
    <S extends T> List<S> saveAll(Iterable<S>);
    List<T> findAll();
    List<T> findAllById(Iterable<ID>);
}
```

315.1. CrudRepository save() New

We can use the `CrudRepository.save()` method to either create or update our `@Entity` instance in the database.

In this specific example, we call `save()` with a new object. The JPA provider can tell this is a new object because the generated primary key value is currently unassigned. An object type has a default value of null in Java. Our primitive `int` type has a default value of 0 in Java.

`CrudRepository.save() New Example`

```
//given an entity instance
Song song = mapper.map(dtoFactory.make());
assertThat(song.getId()).isZero(); ①
//when persisting
songsRepo.save(song);
//then entity is persisted
then(song.getId()).isNotZero(); ②
```

① default value for generated primary key using primitive type interpreted as unassigned

② primary key assigned by provider

The following shows the SQL that is generated by JPA provider to add the new object to the database.

CrudRepository.save() New Example SQL

```
select next value for reposongs_song_sequence
insert into reposongs_song (artist, released, title, id) values (?, ?, ?, ?)
```

315.2. CrudRepository save() Update Existing

The `CrudRepository.save()` method is an "upsert".

- if the `@Entity` is new, the repository will call `EntityManager.persist` as you saw in the previous example
- if the `@Entity` exists, the repository will call `EntityManager.merge` to update the database

CrudRepository.save() Update Existing Example

```
//given an entity instance
Song song = mapper.map(dtoFactory.make());
songsRepo.save(song);
songsRepo.flush(); //for demo only ①
Song updatedSong = Song.builder()
    .id(song.getId()) ③
    .title("new title")
    .artist(song.getArtist())
    .released(song.getReleased())
    .build(); ②
//when persisting update
songsRepo.save(updatedSong);
//then entity is persisted
then(songsRepo.findOne(Example.of(updatedSong))).isPresent(); ④
```

① making sure `@Entity` has been saved

② a new, unmanaged `@Entity` instance is created for a fresh update of the database

③ new, unmanaged `@Entity` instance has an assigned, non-default primary key value

④ object's new state is found in the database

315.3. CrudRepository save()/Update Resulting SQL

The following snippet shows the SQL executed by the repository/EntityManager during the `save()`—where it must first determine if the object exists in the database before calling SQL `INSERT` or `UPDATE`.

CrudRepository.save() Update Existing Example SQL

```
select ... ①
from reposongs_song song0_
where song0_.id=?
binding parameter [1] as [INTEGER] - [1]

update reposongs_song set artist=?, released=?, title=? where id=? ②
binding parameter [1] as [VARCHAR] - [The Beach Boys]
binding parameter [2] as [DATE] - [2010-06-07]
binding parameter [3] as [VARCHAR] - [new title]
binding parameter [4] as [INTEGER] - [1]
```

① EntityManager.merge() performs `SELECT` to determine if assigned primary key exists and loads that state

② EntityManager.merge() performs `UPDATE` to modify state of existing @Entity in database

315.4. New Entity?

We just saw where the same method (`save()`) was used to both create or update the object in the database. This works differently depending on how the repository can determine whether the `@Entity` instance passed to it is new or not.

- for auto-assigned primary keys, the `@Entity` instance is considered new if `@Version` (not used in our example) and `@Id` are not assigned — as long as the `@Id` type is non-primitive.
- for manually assigned and primitive `@Id` types, `@Entity` can implement the `Persistable<ID>` interface to assist the repository in knowing when the `@Entity` is new.

Persistable<ID> Interface

```
public interface Persistable<ID> {
    @Nullable
    ID getId();
    boolean isNew();
}
```

315.5. CrudRepository.existsById()

Spring Data JPA adds a convenience method that can check whether the `@Entity` exists in the database without loading the entire object or writing a custom query.

The following snippet demonstrates how we can check for the existence of a given ID.

CrudRepository.existsById()

```
//given a persisted entity instance
Song pojoSong = mapper.map(dtoFactory.make());
songsRepo.save(pojoSong);
```

```
//when - determining if entity exists
boolean exists = songsRepo.existsById(pojoSong.getId());
//then
then(exists).isTrue();
```

The following shows the SQL produced from the `findById()` call.

CrudRepository.existsById() SQL

```
select count(*) from reposongs_song s1_0 where s1_0.id=? ①
```

① `count(*)` avoids having to return all column values

315.6. CrudRepository findById()

If we need the full object, we can always invoke the `findById()` method, which should be a thin wrapper above `EntityManager.find()`, except that the return type is a Java `Optional<T>` versus the `@Entity` type (`T`).

CrudRepository.findById()

```
//when - finding the existing entity
Optional<Song> result = songsRepo.findById(pojoSong.getId());
//then
then(result).isPresent(); ①
```

① `findById()` always returns a non-null `Optional<T>` object

315.6.1. CrudRepository findById() Found Example

The `Optional<T>` can be safely tested for existence using `isPresent()`. If `isPresent()` returns `true`, then `get()` can be called to obtain the targeted `@Entity`.

Present Optional Example

```
//given
then(result).isPresent();
//when - obtaining the instance
Song dbSong = result.get();
//then - instance provided
then(dbSong).isNotNull();
```

315.6.2. CrudRepository findById() Not Found Example

If `isPresent()` returns `false`, then `get()` will throw a `NoSuchElementException` if called. This gives your code some flexibility for how you wish to handle a target `@Entity` not being found.

Missing Optional Example

```
//given
then(result).isNotPresent();
//then - the optional is asserted during the get()
assertThatThrownBy(() -> result.get())
    .isInstanceOf(NoSuchElementException.class);
```

315.7. CrudRepository delete()

The repository also offers a wrapper around `EntityManager.delete()` where an instance is required. Whether the instance existed or not, a successful call will always result in the `@Entity` no longer in the database.

CrudRepository delete() Example

```
//when - deleting an existing instance
songsRepo.delete(existingSong);
//then - instance will be removed from DB
then(songsRepo.existsById(existingSong.getId())).isFalse();
```

315.7.1. CrudRepository delete() Not Loaded

However, if the instance passed to the `delete()` method is not in its current Persistence Context, then it will load it before deleting so that it has all information required to implement any JPA delete cascade events.

CrudRepository delete() Exists Example SQL

```
select ... from reposongs_song s1_0 where s1_0.id=? ①
delete from reposongs_song where id=?
```

① `@Entity` loaded as part of implementing a delete

JPA Supports Cascade Actions



JPA relationships can be configured to perform an action (e.g., delete) to both sides of the relationship when one side is acted upon (e.g., deleted). This could allow a parent `Album` to be persisted, updated, or deleted with all of its child `Songs` with a single call to the repository/`EntityManager`.

315.7.2. CrudRepository delete() Not Exist

If the instance did not exist, the `delete()` call silently returns.

CrudRepository delete() Does Not Exists Example

```
//when - deleting a non-existing instance
```

```
songsRepo.delete(doesNotExist); ①
```

① no exception thrown for not exist

CrudRepository delete() Does Not Exists Example SQL

```
select ... from reposongs_song s1_0 where s1_0.id=? ①
```

① no `@Entity` was found/loaded as a result of this call

315.8. CrudRepository deleteById()

Spring Data JPA also offers a convenience `deleteById()` method taking only the primary key.

CrudRepository deleteById() Example

```
//when - deleting an existing instance  
songsRepo.deleteById(existingSong.getId());
```

However, since this is JPA under the hood and JPA may have cascade actions defined, the `@Entity` is still retrieved if it is not currently loaded in the Persistence Context.

CrudRepository deleteById() Example SQL

```
select ... from reposongs_song s1_0 where s1_0.id=?  
delete from reposongs_song where id=?
```

deleteById will Throw Exception

Calling `deleteById` for a non-existent `@Entity` will



- throw a `EmptyResultDataAccessException` <= Spring Boot 3.0.6
- quietly return >= Spring Boot 3.1.0

315.9. Other CrudRepository Methods

That was a quick tour of the `CrudRepository<T, ID>` interface methods. The following snippet shows the methods not covered. Most additional `CrudRepository` methods provide convenience methods around the entire repository. The `ListCrudRepository` override the `Iterable<T>` return type with `List<T>`.

Other CrudRepository Methods

```
//public interface CrudRepository<T, ID> extends Repository<T, ID> {  
<S extends T> Iterable<S> saveAll(Iterable<S>);  
Iterable<T> findAll();  
Iterable<T> findAllById(Iterable<ID>);  
long count();
```

```
void deleteAll(Iterable<? extends T>);  
void deleteAll();  
  
//public interface ListCrudRepository<T, ID> extends CrudRepository<T, ID> {  
<S extends T> List<S> saveAll(Iterable<S>);  
List<T> findAll();  
List<T> findAllById(Iterable<ID>);
```

Chapter 316. PagingAndSortingRepository

Before we get too deep into queries, it is good to know that Spring Data has first-class support for sorting and paging.

- **sorting** - determines the order which matching results are returned
- **paging** - breaks up results into chunks that are easier to handle than entire database collections

Here is a look at the declared methods of the `PagingAndSortingRepository<T, ID>` interface and the `ListPagingAndSortingRepository<T, ID>` overrides. These interfaces define `findAll()` methods that accept paging and sorting parameters and return types. They define extra parameters not included in the `CrudRepository` `findAll()` methods.

PagingAndSortingRepository<T, ID> Interface

```
public interface PagingAndSortingRepository<T, ID> extends Repository<T, ID> {  
    Iterable<T> findAll(Sort);  
    Page<T> findAll(Pageable);  
}  
  
public interface ListPagingAndSortingRepository<T, ID> extends  
PagingAndSortingRepository<T, ID> {  
    List<T> findAll(Sort);  
}
```

We will see the paging and sorting option come up in many other query types as well.



Use Paging and Sorting for Collection Queries

All queries that return a collection should seriously consider adding paging and sorting parameters. Small test databases can become significantly populated production databases over time and cause eventual failure if paging and sorting are not applied to unbounded collection query return methods.

316.1. Sorting

Sorting can be performed on one or more properties and in ascending and descending order.

The following snippet shows an example of calling the `findAll()` method and having it return

- `Song` entities in descending order according to `release` date
- `Song` entities in ascending order according to `id` value when `release` dates are equal

Sort.by() Example

```
//when  
List<Song> byReleased = songsRepository.findAll(  
    Sort.by("released").descending().and(Sort.by("id").ascending())); ① ②  
//then
```

```

LocalDate previous = null;
for (Song s: byReleased) {
    if (previous!=null) {
        then(previous).isAfterOrEqualTo(s.getReleased()); //DESC order
    }
    previous=s.getReleased();
}

```

- ① results can be sorted by one or more properties
- ② order of sorting can be ascending or descending

The following snippet shows how the SQL was impacted by the `Sort.by()` parameter.

Sort.by() Example SQL

```

select ...
from reposongs_song s1_0
order by s1_0.released desc,s1_0.id ①

```

- ① `Sort.by()` added the extra SQL `order by` clause

316.2. Paging

Paging permits the caller to designate how many instances are to be returned in a call and the offset to start that group (called a page or slice) of instances.

The snippet below shows an example of using one of the factory methods of `Pageable` to create a `PageRequest` definition using page size (limit), offset, and sorting criteria. If many pages are traversed—it is advised to sort by a property that will produce a stable sort over time during table modifications.

Defining Initial Pageable

```

//given
int offset = 0;
int pageSize = 3;
Pageable pageable = PageRequest.of(offset/pageSize, pageSize, Sort.by("released"));①
②
//when
Page<Song> songPage = songsRepository.findAll(pageable);

```

- ① using `PageRequest` factory method to create `Pageable` from provided page information
- ② parameters are pageNumber, pageSize, and Sort

Use Stable Sort over Large Collections



Try to use a property for sort (at least by default) that will produce a stable sort when paging through a large collection to avoid repeated or missing objects from follow-on pages because of new changes to the table.

316.3. Page Result

The page result is represented by a container object of type `Page<T>`, which extends `Slice<T>`. I will describe the difference next, but the `PagingAndSortingRepository<T, ID>` interface always returns a `Page<T>`, which will provide:

- the sequential number of the page/slice
- the requested size of the page/slice
- the number of elements found
- the total number of elements available in the database

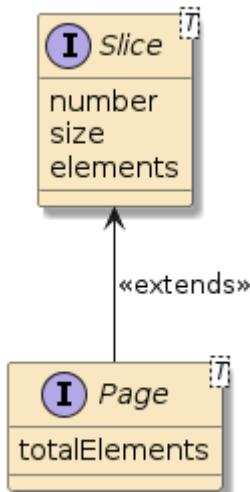


Figure 136. `Page<T>` Extends `Slice<T>`

Page Issues Extra Count Query



Of course, the total number of elements available in the database does not come for free. An extra query is performed to get the count. If that attribute is unnecessary, use a Slice return using a derived query.

316.4. Slice Properties

The `Slice<T>` base interface represents properties about the content returned.

Slice Properties

```
//then
Slice songSlice = songPage; ①
then(songSlice).isNotNull();
then(songSlice.isEmpty()).toBeFalsy();
then(songSlice.getNumber()).isEqualTo(0); ②
then(songSlice.getSize()).isEqualTo(pageSize); ③
then(songSlice.getNumberofElements()).isEqualTo(pageSize); ④

List<Song> songsList = songSlice.getContent();
then(songsList).hasSize(pageSize);
```

① `Page<T>` extends `Slice<T>`

② slice increment — first slice is 0

③ the number of elements requested for this slice

- ④ the number of elements returned in this slice

316.5. Page Properties

The `Page<T>` derived interface represents properties about the entire collection/table.

The snippet below shows an example of the total number of elements in the table being made available to the caller.

Page Properties

```
then(songPage.getTotalElements()).isEqualTo(savedSongs.size()); //unique to Page
```

The `Page<T>` content and number of elements is made available through the following set of SQL queries.

Page Resulting SQL

```
select ... from reposongs_song s1_0 ①
  order by s1_0.released
  offset ? rows fetch first ? rows only

select count(s1_0.id) from reposongs_song s1_0 ②
```

① `SELECT` used to load page of entities (aka the `Slice` information)

② `SELECT COUNT(*)` used to return total matches in the database—returned or not because of `Pageable` limits (aka the `Page` portion of the information)

316.6. Stateful Pageable Creation

In the above example, we created a `Pageable` from stateless parameters—passing in `pageNumber`, `pageSize`, and sorting specifications.

Review: Stateless Pageable Definition

```
Pageable pageable = PageRequest.of(offset / pageSize, pageSize, Sort.by("released"))
;①
```

① parameters are `pageNumber`, `pageSize`, and `Sort`

We can also use the original `Pageable` to generate the next or other relative page specifications.

Relative Pageable Creation

```
Pageable next = pageable.next();
Pageable previous = pageable.previousOrFirst();
Pageable first = pageable.first();
```

316.7. Page Iteration

The next `Pageable` can be used to advance through the complete set of query results, using the previous `Pageable` and testing the returned `Slice`.

Page Iteration

```
for (int i=1; songSlice.hasNext(); i++) { ①
    pageable = pageable.next(); ②
    songSlice = songsRepository.findAll(pageable);
    songsList = songSlice.getContent();
    then(songSlice).isNotNull();
    then(songSlice.getNumber()).isEqualTo(i);
    then(songSlice.getSize()).isEqualTo(pageSize);
    then(songSlice.getNumberOfElements()).isLessThanOrEqualTo(pageSize);
    then(((Page)songSlice).getTotalElements()).isEqualTo(savedSongs.size());//unique
    to Page
}
then(songSlice.hasNext()).isFalse();
then(songSlice.getNumber()).isEqualTo(songsRepository.count() / pageSize);
```

① `Slice.hasNext()` will indicate when previous `Slice` represented the end of the results

② next `Pageable` obtained from previous `Pageable`

The following snippet shows an example of the SQL issued to the database with each page request.

Page Iteration SQL

```
select ... from reposongs_song s1_0
    order by s1_0.released
    offset ? rows fetch first ? rows only
--binding parameter [1] as [INTEGER] - [6] ①
--binding parameter [2] as [INTEGER] - [3]

select count(s1_0.id) from reposongs_song s1_0
```

① paging advances offset

Chapter 317. Query By Example

Not all queries will be as simple as `findAll()`. We now need to start looking at queries that can return a subset of results based on them matching a set of predicates. The `QueryByExampleExecutor<T>` parent interface to `JpaRepository<T, ID>` provides a set of variants to the collection-based results that accepts an "example" to base a set of predicates off of.

QueryByExampleExecutor<T> Interface

```
public interface QueryByExampleExecutor<T> {  
    <S extends T> Optional<S> findOne(Example<S>);  
    <S extends T> Iterable<S> findAll(Example<S>);  
    <S extends T> Iterable<S> findAll(Example<S>, Sort);  
    <S extends T> Page<S> findAll(Example<S>, Pageable);  
    <S extends T> long count(Example<S>);  
    <S extends T> boolean exists(Example<S>);  
    <S extends T, R> R findBy(Example<S>, Function<FluentQuery$FetchableFluentQuery<S>, R>);  
}
```

317.1. Example Object

An `Example` is an interface with the ability to hold onto a probe and matcher.

317.1.1. Probe Object

The probe is an instance of the repository `@Entity` type.

The following snippet is an example of creating a probe that represents the fields we are looking to match.

Probe Example

```
//given  
Song savedSong = savedSongs.get(0);  
Song probe = Song.builder()  
    .title(savedSong.getTitle())  
    .artist(savedSong.getArtist())  
    .build(); ①
```

① probe will carry values for `title` and `artist` to match

317.1.2. ExampleMatcher Object

The matcher defaults to an exact match of all non-null properties in the probe. There are many definitions we can supply to customize the matcher.

- `ExampleMatcher.matchingAny()` - forms an OR relationship between all predicates

- `ExampleMatcher.matchingAll()` - forms an AND relationship between all predicates

The `matcher` can be broken down into specific fields, designing a fair number of options for String-based predicates but very limited options for non-String fields.

- exact match
- case-insensitive match
- starts with, ends with
- contains
- regular expression
- include or ignore nulls

The following snippet shows an example of the default `ExampleMatcher`.

Default ExampleMatcher

```
ExampleMatcher matcher = ExampleMatcher.matching(); ①
```

① default matcher is `matchingAll`

317.2. findAll By Example

We can supply an `Example` instance to the `findAll()` method to conduct our query.

The following snippet shows an example of using a probe with a default matcher. It is intended to locate all songs matching the `artist` and `title` we specified in the probe.

```
//when
List<Song> foundSongs = songsRepository.findAll(
    Example.of(probe), //default matcher is matchingAll() and non-null
    Sort.by("id"));
```

However, there is a problem. Our `Example` instance with supplied probe and default matcher did not locate any matches.

No Matches Found - huh?

```
//then - not found
then(foundSongs).isEmpty();
```

317.3. Primitive Types are Non-Null

The reason for the no-match is that the primary key value is being added to the query, and we did not explicitly supply that value in our probe.

No Matches SQL

```
select ... from reposongs_song s1_0
where s1_0.id=? --filled in with 0 ①
and s1_0.artist=? and s1_0.title=?
```

```

order by s1_0.id
--binding parameter [1] as [INTEGER] - [0]
--binding parameter [2] as [VARCHAR] - [Creedence Clearwater Revival]
--binding parameter [3] as [VARCHAR] - [Quo Vadis green]

```

① `id=0` test for unassigned primary key, prevents match being found

The `id` field is a primitive `int` type that cannot be null and defaults to a 0 value. That, and the fact that the default matcher is a "match all" (using `AND`) keeps our example from matching anything.

@Entity Uses Primitive Type for Primary Key

```

@Entity
public class Song {
    @Id @GeneratedValue
    private int id; ①
}

```

① `id` can never be null and defaults to 0, unassigned value

317.4. matchingAny ExampleMatcher

One option we could take would be to switch from the default `matchingAll` matcher to a `matchingAny` matcher.

The following snippet shows an example of how we can specify the override.

matchingAny ExampleMatcher Example

```

//when
List<Song> foundSongs = songsRepository.findAll(
    Example.of(probe, ExampleMatcher.matchingAny()), ①
    Sort.by("id"));

```

① using `matchingAny` versus default `matchingAll`

This causes some matches to occur, but it likely is not what we want.

- the `id` predicate is still being supplied
- the overall condition does not require the `artist AND title` to match.

matchingAny ExampleMatcher Example SQL

```

select ...
from reposongs_song s1_0
where s1_0.id=? --filled in with 0 ①
    or s1_0.artist=? or s1_0.title=?
order by s1_0.id

```

① matching any ("or") of the non-null probe values

317.5. Ignoring Properties

What we want to do is use a `matchAll` matcher and have the non-null primitive `id` field ignored.

The following snippet shows an example matcher configured to ignore the primary key.

matchingAll ExampleMatcher with Ignored Property

```
ExampleMatcher ignoreId = ExampleMatcher.matchingAll().withIgnorePaths("id");①
//when
List<Song> foundSongs = songsRepository.findAll(
    Example.of(probe, ignoreId), ②
    Sort.by("id"));
//then
then(foundSongs).isNotEmpty();
then(foundSongs.get(0).getId()).isEqualTo(savedSong.getId());
```

① `id` primary key is being excluded from predicates

② non-null and non-id fields of probe are used for **AND** matching

The following snippet shows the SQL produced. This SQL matches only the `title` and `artist` fields, without a reference to the `id` field.

matchingAll ExampleMatcher with Ignored Property SQL

```
select ...
from reposongs_song s1_0
where s1_0.artist=? and s1_0.title=? ① ②
order by s1_0.id
```

① the primitive `int id` field is being ignored

② both `title` and `artist` fields must match

317.6. Contains ExampleMatcher

We have some options on what we can do with the String matches.

The following snippet provides an example of testing whether `title` contains the text in the probe while performing an exact match of the `artist` and ignoring the `id` field.

Contains ExampleMatcher

```
Song probe = Song.builder()
    .title(savedSong.getTitle().substring(2))
    .artist(savedSong.getArtist())
    .build();
ExampleMatcher matcher = ExampleMatcher
    .matching()
    .withIgnorePaths("id")
```

```
.withMatcher("title", ExampleMatcher.GenericPropertyMatchers.contains());
```

317.6.1. Using Contains ExampleMatcher

The following snippet shows that the `Example` successfully matched on the `Song` we were interested in.

Example is Found

```
//when
List<Song> foundSongs = songsRepository.findAll(Example.of(probe,matcher), Sort.by("id"));
//then
then(foundSongs).isNotEmpty();
then(foundSongs.get(0).getId()).isEqualTo(savedSong.getId());
```

The following SQL shows what was performed by our `Example`. Both `title` and `artist` are required to match. The match for `title` is implemented as a "contains" `LIKE`.

Contains Example SQL

```
select ...
from reposongs_song s1_0
where s1_0.artist=? and s1_0.title like ? escape '\' ②
order by s1_0.id
//binding parameter [1] as [VARCHAR] - [Earth Wind and Fire]
//binding parameter [2] as [VARCHAR] - [% a God Unknown%] ①
```

① title parameter supplied with % characters around the probe value

② title predicate uses a LIKE

Chapter 318. Derived Queries

For fairly straight forward queries, Spring Data JPA can derive the required commands from a method signature declared in the repository interface. This provides a more self-documenting version of similar queries we could have formed with query-by-example.

The following snippet shows a few example queries added to our repository interface to address specific queries needed in our application.

Example Query Method Names

```
public interface SongsRepository extends JpaRepository<Song, Integer> {  
    Optional<Song> getByTitle(String title); ①  
  
    List<Song> findByTitleNullAndReleasedAfter(LocalDate date); ②  
  
    List<Song> findByTitleStartingWith(String string, Sort sort); ③  
    Slice<Song> findByTitleStartingWith(String string, Pageable pageable); ④  
    Page<Song> findPageByTitleStartingWith(String string, Pageable pageable); ⑤
```

- ① query by an exact match of `title`
- ② query by a match of two fields
- ③ query using sort
- ④ query with paging support
- ⑤ query with paging support and table total

Let's look at a complete example first.

318.1. Single Field Exact Match Example

In the following example, we have created a query method `getByTitle` that accepts the exact match title value and an `Optional` return value.

Interface Method Signature

```
Optional<Song> getByTitle(String title); ①
```

We use the declared interface method normally, and Spring Data JPA takes care of the implementation.

Interface Method Usage

```
//when  
Optional<Song> result = songsRepository.getByTitle(song.getTitle());  
//then  
then(result).isPresent();
```

The resulting SQL is the same as if we implemented it using query-by-example or JPA query language.

Resulting SQL

```
select ...
from reposongs_song s1_0
where s1_0.title=?
```

318.2. Query Keywords

Spring Data has several **keywords**, followed by **By**, that it looks for starting the interface method name. Those with multiple terms can be used interchangeably.

Meaning	Keywords
Query	<ul style="list-style-type: none">• find• read• get• query• search• stream
Count	<ul style="list-style-type: none">• count
Exists	<ul style="list-style-type: none">• exists
Delete	<ul style="list-style-type: none">• delete• remove

318.3. Other Keywords

Other keywords include [\[1\]](#) [\[2\]](#)

- Distinct (e.g., `findDistinctByTitle`)
- Is, Equals (e.g., `findByTitle`, `findByTitleIs`, `findByTitleEquals`)
- Not (e.g., `findByTitleNot`, `findByTitleIsNot`, `findByTitleNotEquals`)
- IsNull, IsNotNull (e.g., `findByTitle(null)`, `findByTitleIsNull()`, `findByTitleIsNotNull()`)
- StartingWith, EndingWith, Containing (e.g., `findByTitleStartingWith`, `findByTitleEndingWith`, `findByTitleContaining`)
- LessThan, LessThanEqual, GreaterThan, GreaterThanEqual, Between (e.g., `findByIdLessThan`, `findByIdBetween(lo,hi)`)
- Before, After (e.g., `findByReleaseAfter`)
- In (e.g., `findByTitleIn(collection)`)
- OrderBy (e.g., `findByTitleContainingOrderByTitle`)

The list is significant but not meant to be exhaustive. Perform a web search for your specific needs (e.g., "Spring Data Derived Query ...") if what is needed is not found here.

318.4. Multiple Fields

We can define queries using one or more fields using **And** and **Or**.

The following example defines an interface method that will test two fields: **title** and **released**. **title** will be tested for null and **released** must be after a certain date.

Multiple Fields Interface Method Declaration

```
List<Song> findByTitleNullAndReleasedAfter(LocalDate date);
```

The following snippet shows an example of how we can call/use the repository method. We are using a simple collection return without sorting or paging.

Multiple Fields Example Use

```
//when
List<Song> foundSongs = songsRepository.findByTitleNullAndReleasedAfter(firstSong
    .getReleased());
//then
Set<Integer> foundIds = foundSongs.stream()
    .map(s->s.getId())
    .collect(Collectors.toSet());
then(foundIds).isEqualTo(expectedIds);
```

The resulting SQL shows that a query is performed looking for null **title** and **released** after the **LocalDate** provided.

Multiple Fields Resulting SQL

```
select ...
from reposongs_song s1_0
where s1_0.title is null and s1_0.released >?
```

318.5. Collection Response Query Example

We can perform queries with various types of additional arguments and return types. The following shows an example of a query that accepts a sorting order and returns a simple collection with all objects found.

Collection Response Interface Method Declaration

```
List<Song> findByTitleStartingWith(String string, Sort sort);
```

The following snippet shows an example of how to form the **Sort** and call the query method derived from our interface declaration.

Collection Response Interface Method Use

```
//when
Sort sort = Sort.by("id").ascending();
List<Song> songs = songsRepository.findByTitleStartingWith(startingWith, sort);
//then
then(songs.size()).isEqualTo(expectedCount);
```

The following shows the resulting SQL—which now contains a sort clause based on our provided definition.

Collection Response Resulting SQL

```
select ...
from reposongs_song s1_0
where s1_0.title like ? escape \
order by s1_0.id
```

318.6. Slice Response Query Example

Derived queries can also be declared to accept a `Pageable` definition and return a `Slice`. The following example shows a similar interface method declaration to what we had prior—except we have wrapped the `Sort` within a `Pageable` and requested a `Slice`, which will contain only those items that match the predicate and comply with the paging constraints.

Slice Response Interface Method Declaration

```
Slice<Song> findByTitleStartingWith(String string, Pageable pageable);
```

The following snippet shows an example of forming the `PageRequest`, making the call, and inspecting the returned `Slice`.

Slice Response Interface Method Use

```
//when
PageRequest pageable=PageRequest.of(0, 1, Sort.by("id").ascending());
Slice<Song> songsSlice=songsRepository.findByTitleStartingWith(startingWith, pageable
);
//then
then(songsSlice.getNumberElements()).isEqualTo(pageable.getPageSize());
```

The following resulting SQL shows how paging offset and limits were placed in the query.

Slice Response Resulting SQL

```
select ...
from reposongs_song s1_0
where s1_0.title like ? escape \ $\backslash$ 
```

```
order by s1_0.id  
offset ? rows fetch first ? rows only  
--binding parameter [1] as [VARCHAR] - [F%]  
--binding parameter [2] as [INTEGER] - [0]  
--binding parameter [3] as [INTEGER] - [2]
```

318.7. Page Response Query Example

We can alternatively declare a `Page` return type if we also need to know information about all available matches in the table. The following shows an example of returning a `Page`. The only reason `Page` shows up in the method name is to form a different method signature than its sibling examples. `Page` is not required to be in the method name.

Page Response Interface Method Declaration

```
Page<Song> findPageByTitleStartingWith(String string, Pageable pageable);
```

The following snippet shows how we can form a `PageRequest` to pass to the derived query method and accept a `Page` in response with additional table information.

Page Response Interface Method Use

```
//when  
PageRequest pageable = PageRequest.of(0, 1, Sort.by("id").ascending());  
Page<Song> songsPage = songsRepository.findPageByTitleStartingWith(startingWith,  
pageable);  
//then  
then(songsPage.getNumberElements()).isEqualTo(pageable.getPageSize());  
then(songsPage.getTotalElements()).isEqualTo(expectedCount); ①
```

① an extra property is available to tell us the total number of matches relative to the entire table—that may not have been reported on the current page

The following shows the resulting SQL of the `Page` response. Note that two queries were performed. One provided all the data required for the parent `Slice` and the second query provided the table totals not bounded by the page limits.

Page Response Resulting SQL

```
select ... ①  
from reposongs_song s1_0  
where s1_0.title like ? escape '\'  
order by s1_0.id  
offset ? rows fetch first ? rows only  
--binding parameter [1] as [VARCHAR] - [T%]  
--binding parameter [2] as [INTEGER] - [0]  
--binding parameter [3] as [INTEGER] - [1]  
  
select count(s1_0.id) ②
```

```
from reposongs_song s1_0
where s1_0.title like ? escape '\'
--binding parameter [1] as [VARCHAR] - [T%]
```

- ① first query provides **Slice** data within **Pageable** limits (offset omitted for first page)
- ② second query provides table-level count for **Page** that have no page size limits

[1] *"Query Creation"*, Spring Data JPA - Reference Documentation

[2] *"Derived Query Methods in Spring Data JPA"*, Atta

Chapter 319. JPA-QL Named Queries

Query-by-example and derived queries are targeted at flexible but mostly simple queries. Often there is a need to write more complex queries. If you remember in JPA, we can write JPA-QL and native SQL queries to implement our database query access. We can also register them as a `@NamedQuery` associated with the `@Entity` class. This allows for more complex queries as well as to use queries defined in a JPA `orm.xml` source file (without having to recompile)

The following snippet shows a `@NamedQuery` called `Song.findArtistGESize` that implements a query of the `Song` entity's table to return `Song` instances that have artist names longer than a particular size.

JPA-QL @NamedQuery Can Express More Complex Queries

```
@Entity  
 @Table(name="REPOSONGS_SONG")  
 @NamedQuery(name="Song.findByArtistGESize",  
             query="select s from Song s where length(s.artist) >= :length")  
 public class Song {
```

The following snippet shows an example of using that `@NamedQuery` with the JPA `EntityManager`.

JPA Named Query Syntax

```
TypedQuery<Song> query = entityManager  
    .createNamedQuery("Song.findByArtistGESize", Song.class)  
    .setParameter("length", minLength);  
List<Song> jpaFoundSongs = query.getResultList();
```

319.1. Mapping @NamedQueries to Repository Methods

That same tool is still available to us with repositories. If we name the query `[prefix].[suffix]`, where `prefix` is the `@Entity.name` of the objects returned and `suffix` matches the name of the repository interface method—we can have them automatically called by our repository.

The following snippet shows a repository interface method that will have its query defined by the `@NamedQuery` defined on the `@Entity` class. Note that we map repository method parameters to the `@NamedQuery` parameter using the `@Param` annotation.

Repository Interface Methods can Automatically Invoke Matching @NamedQueries

```
//see @NamedQuery(name="Song.findByArtistGESize" in Song class  
List<Song> findByArtistGESize(@Param("length") int length); ① ②
```

① interface method name matches `@NamedQuery.name` suffix

② `@Param` maps method parameter to `@NamedQuery` parameter

The following snippet shows the resulting SQL generated from the JPA-QL/@NamedQuery

JPA-QL Resulting SQL

```
select ...
from reposongs_song s1_0
where character_length(s1_0.artist)>=?
```

Chapter 320. @Query Annotation Queries

Spring Data JPA provides an option for the query to be expressed on the repository method versus the `@Entity` class.

The following snippet shows an example of a similar query we did for `artist` length—except in this case we are querying against `title` length.

Query Supplied on Repository Method

```
@Query("select s from Song s where length(s.title) >= :length")
List<Song> findByTitleGESize(@Param("length") int length);
```

We get the expected resulting SQL.

Resulting SQL

```
select ...
from reposongs_song s1_0
where character_length(s1_0.artist)>=?
```

Named Queries can be supplied in property file

Named queries can also be expressed in a property file—versus being placed directly onto the method. Property files can provide a more convenient source for expressing more complex queries.



```
@EnableJpaRepositories(namedQueriesLocation="...")
```

The default location is `META-INF/jpa-named-queries.properties`

320.1. @Query Annotation Native Queries

Although I did not demonstrate it, the `@NamedQuery` can also be expressed in native SQL. In most cases with native SQL queries, the returned information is just data. We can also directly express the repository interface method as a native SQL query as well as have it return straight data.

The following snippet shows a repository interface method implemented as native SQL that will return only the `title` columns based on size.

Example Native SQL @Query Method

```
@Query(value="select s.title from REPOSONGS_SONG s where length(s.title) >= :length",
nativeQuery=true)
List<String> getTitlesGESizeNative(@Param("length") int length);
```

The following output shows the resulting SQL. We can tell this was from a native SQL query

because the SQL does not contain mangled names used by JPA generated SQL.

Resulting Native SQL

```
select s.title ①
from REPOSONGS_SONG s
where length(s.title) >= ?
```

① native SQL query gets expressed exactly as we supplied it

320.2. @Query Sort and Paging

The `@Query` approach supports paging via `Pageable` parameter. Sort must be defined within the query.

@Query Sort and Paging

```
@Query(value="select s from Song s where released between :starting and :ending order
by id ASC")
Page<Song> findByReleasedBetween(LocalDate starting, LocalDate ending, Pageable
pageable);
```

Chapter 321. JpaRepository Methods

Many of the methods and capabilities of the `JpaRepository<T, ID>` are available at the higher level interfaces. The `JpaRepository<T, ID>` itself declares four types of additional methods

- flush-based methods
- batch-based deletes
- reference-based accessors
- return type extensions

JpaRepository<T, ID> Interface

```
public interface JpaRepository<T, ID> extends ListCrudRepository<T, ID>,
ListPagingAndSortingRepository<T, ID>, QueryByExampleExecutor<T> {
    void flush();
    <S extends T> S saveAndFlush(S);
    <S extends T> List<S> saveAllAndFlush(Iterable<S>);

    void deleteAllInBatch(Iterable<T> entities);
    void deleteAllByIdInBatch(Iterable<ID>);
    void deleteAllInBatch();

    T getReferenceById(ID);

    <S extends T> List<S> findAll(Example<S>);
    <S extends T> List<S> findAll(Example<S>, Sort);
}
```

321.1. JpaRepository Type Extensions

The methods in the `JpaRepository<T, ID>` interface not discussed here mostly just extend existing parent methods with more concrete return types (e.g., `List` versus `Iterable`).

Abstract Generic Spring Data Methods

```
public interface QueryByExampleExecutor<T> {
    <S extends T> Iterable<S> findAll(Example<S> example);
```

Concrete Spring Data JPA Extensions

```
public interface JpaRepository<T, ID> extends ..., QueryByExampleExecutor<T> {
    @Override
    <S extends T> List<S> findAll(Example<S> example); ①
    ...
}
```

① `List<T>` extends `Iterable<T>`

321.2. JpaRepository flush()

As we know with JPA, many commands are cached within the local Persistence Context and issued to the database at some point in time in the future. That point in time is either the end of the transaction or some event within the scope of the transaction (e.g., issue a JPA query). `flush()` commands can be used to immediately force queued commands to the database. We would need to do this before issuing a native SQL command if we want our latest changes to be included with that command.

In the following example, a transaction is held open during the entire method because of the `@Transactional` declaration. `saveAll()` just adds the objects to the Persistence Context and caches their insert commands. The `flush()` command finally forces the SQL `INSERT` commands to be issued.

```
@Test
@Transactional
void flush() {
    //given
    List<Song> songs = dtoFactory.listBuilder().songs(5,5).stream()
        .map(s->mapper.map(s))
        .collect(Collectors.toList());
    songsRepository.saveAll(songs); ①
    //when
    songsRepository.flush(); ②
}
```

① instances are added to the Persistence Unit cache

② instances are explicitly flushed to the database

The pre-flush actions are only to assign the primary key value.

Database Calls Pre-Flush

```
Hibernate: select next value for reposongs_song_sequence
Hibernate: select next value for reposongs_song_sequence
```

The post-flush actions insert the rows into the database.

Database Calls Post-Flush

```
Hibernate: insert into reposongs_song (artist, released, title, id) values (?, ?, ?, ?, ?)
Hibernate: insert into reposongs_song (artist, released, title, id) values (?, ?, ?, ?, ?)
Hibernate: insert into reposongs_song (artist, released, title, id) values (?, ?, ?, ?, ?)
Hibernate: insert into reposongs_song (artist, released, title, id) values (?, ?, ?, ?, ?)
Hibernate: insert into reposongs_song (artist, released, title, id) values (?, ?, ?, ?, ?)
```



Call flush() Before Issuing Native SQL Queries

You do not need to call `flush()` in order to eventually have changes written to the

database. However, you must call `flush()` within a transaction to assure that all changes are available to native SQL queries issued against the database. JPA-QL queries will automatically call `flush()` before executing.

321.3. JpaRepository deleteInBatch

The standard `deleteAll(collection)` will issue deletes one SQL statement at a time as shown in the comments of the following snippet.

```
songsRepository.deleteAll(savedSongs);
//delete from reposongs_song where id=? ①
//delete from reposongs_song where id=? 
//delete from reposongs_song where id=?
```

① SQL `DELETE` commands are issued one at a time for each ID

The `JpaRepository.deleteInBatch(collection)` will issue a single DELETE SQL statement with all IDs expressed in the where clause.

```
songsRepository.deleteInBatch(savedSongs);
//delete from reposongs_song where id=? or id=? or id=? ①
```

① one SQL `DELETE` command is issued for all IDs

321.4. JPA References

JPA has the notion of references that represent a promise to an `@Entity` in the database. This is normally done to make loading targeted objects from the database faster and leaving related objects to be accessed only on-demand.

In the following examples, the code is demonstrating how it can form a reference to a persisted object in the database — without going through the overhead of realizing that object.

321.4.1. Reference Exists

In this first example, the referenced object exists and the transaction stays open from the time the reference is created — until the reference was resolved.

Able to Obtain Object through Reference within Active Transaction

```
@Test
@Transactional
void ref_session() {
    ...
    //when - obtaining a reference with a session
    Song dbSongRef = songsRepository.getReferenceById(song.getId()); ①
    //then
    then(dbSongRef).isNotNull();
```

```

    then(dbSongRef.getId()).isEqualTo(song.getId()); ②
    then(dbSongRef.getTitle()).isEqualTo(song.getTitle()); ③
}

```

- ① returns only a reference to the `@Entity` — without loading from the database
- ② still only dealing with the unresolved reference up and to this point
- ③ actual object resolved from the database at this point

321.4.2. Reference Session Inactive

The following example shows that a reference can only be resolved during its initial transaction. We are able to perform some light commands that can be answered directly from the reference, but as soon as we attempt to access data that would require querying the database — it fails.

Unable to Obtain Object through Reference Outside of Transaction

```

import org.hibernate.LazyInitializationException;
...
@Test
void ref_no_session() {
...
    //when - obtaining a reference without a session
    Song dbSongRef = songsRepository.getReferenceById(song.getId()); ①
    //then - get a reference with basics
    then(dbSongRef).isNotNull();
    then(dbSongRef.getId()).isEqualTo(song.getId()); ②
    assertThatThrownBy(
        () -> dbSongRef.getTitle()) ③
        .isInstanceOf(LazyInitializationException.class);
}

```

- ① returns only a reference to the `@Entity` from original transaction
- ② still only dealing with the unresolved reference up and to this point
- ③ actual object resolution attempted at this point — fails

321.4.3. Bogus Reference

The following example shows that the reference is never attempted to be resolved until something is necessary from the object it represents — beyond its primary key.

Reference Never Resolved until Demand

```

import jakarta.persistence.EntityNotFoundException;
...
@Transactional
void ref_not_exist() {
    //given
}

```

```
int doesNotExist=1234;  
//when  
Song dbSongRef = songsRepository.getReferenceById(doesNotExist); ①  
//then - get a reference with basics  
then(dbSongRef).isNotNull();  
then(dbSongRef.getId()).isEqualTo(doesNotExist); ②  
assertThatThrownBy  
    () -> dbSongRef.getTitle()) ③  
    .isInstanceOf(EntityNotFoundException.class);  
}
```

① returns only a reference to the `@Entity` with an ID not in database

② still only dealing with the unresolved reference up and to this point

③ actual object resolution attempted at this point — fails

Chapter 322. Custom Queries

Sooner or later, a repository action requires some complexity beyond the ability to leverage a single query-by-example, derived query, or even JPA-QL. We may need to implement some custom logic or may want to encapsulate multiple calls within a single method.

322.1. Custom Query Interface

The following example shows how we can extend the repository interface to implement custom calls using the JPA `EntityManager` and the other repository methods. Our custom implementation will return a random `Song` from the database.

Interface for Public Custom Query Methods

```
public interface SongsRepositoryCustom {  
    Optional<Song> random();  
}
```

322.2. Repository Extends Custom Query Interface

We then declare the repository to extend the additional custom query interface—making the new method(s) available to callers of the repository.

Repository Implements Custom Query Interface

```
public interface SongsRepository extends JpaRepository<Song, Integer>,  
    SongsRepositoryCustom { ①  
    ...  
}
```

① added additional `SongRepositoryCustom` interface for `SongRepository` to extend

322.3. Custom Query Method Implementation

Of course, the new interface will need an implementation. This will require at least two lower-level database calls

1. determine how many objects there are in the database
2. return a random instance for one of those values

The following snippet shows a portion of the custom method implementation. Note that two additional helper methods are required. We will address them in a moment. By default, this class must have the same name as the interface, followed by "Impl".

Custom Query Method Implementation

```
public class SongsRepositoryCustomImpl implements SongsRepositoryCustom {  
    private final SecureRandom random = new SecureRandom();
```

```

...
    @Override
    public Optional<Song> random() {
        Optional randomSong = Optional.empty();
        int count = (int) songsRepository.count(); ①

        if (count!=0) {
            int offset = random.nextInt(count);
            List<Song> songs = songs(offset, 1); ②
            randomSong = songs.isEmpty() ? Optional.empty():Optional.of(songs.get(0));
        }
        return randomSong;
    }
}

```

① leverages `CrudRepository.count()` helper method

② leverages a local, private helper method to access specific `Song`

322.4. Repository Implementation Postfix

If you have an alternate suffix pattern other than "Impl" in your application, you can set that value in an attribute of the `@EnableJpaRepositories` annotation.

The following shows a declaration that sets the suffix to its normal default value (i.e., we did not have to do this). If we changed this postfix value from "Impl" to "Xxx", then we would need to change `SongsRepositoryCustomImpl` to `SongsRepositoryCustomXxx`.

Optional Custom Query Method Implementation Suffix

```
@EnableJpaRepositories(repositoryImplementationPostfix="Impl") ①
```

① `Impl` is the default value. Configure this attribute to use non-`Impl` postfix

322.5. Helper Methods

The custom `random()` method makes use of two helper methods. One is in the `CrudRepository` interface and the other directly uses the `EntityManager` to issue a query.

CrudRepository.count() Used as Helper Method

```
public interface CrudRepository<T, ID> extends Repository<T, ID> {
    long count();
```

EntityManager NamedQuery used as Helper Method

```
protected List<Song> songs(int offset, int limit) {
    return em.createNamedQuery("Song.songs")
        .setFirstResult(offset)
```

```
.setMaxResults(limit)
.getResultList();
}
```

We will need to inject some additional resources to make these calls:

- `SongsRepository`
- `EntityManager`

322.6. Naive Injections

We could have attempted to inject a `SongsRepository` and `EntityManager` straight into the `Impl` class.

Possible Injection Option

```
@RequiredArgsConstructor
public class SongsRepositoryCustomImpl implements SongsRepositoryCustom {
    private final EntityManager em;
    private final SongsRepository songsRepository;
```

However,

- injecting the `EntityManager` would functionally work, but would not necessarily be part of the same Persistence Context and transaction as the rest of the repository
- eagerly injecting the `SongsRepository` in the `Impl` class will not work because the `Impl` class is now part of the `SongsRepository` implementation. We have a recursion problem to resolve there.

322.7. Required Injections

We need to instead

- inject a `JpaContext` and obtain the `EntityManager` from that context
- use `@Autowired @Lazy` and a non-final attribute for the `SongsRepository` injection to indicate that this instance can be initialized without access to the injected bean

Required Injections

```
import org.springframework.data.jpa.repository.JpaContext;
...
public class SongsRepositoryCustomImpl implements SongsRepositoryCustom {
    private final EntityManager em; ①
    @Autowired @Lazy ②
    private SongsRepository songsRepository;

    public SongsRepositoryCustomImpl(JpaContext jpaContext) { ①
        em=jpaContext.getEntityManagerByManagedType(Song.class);
    }
}
```

- ① EntityManager obtained from injected JpaContext
- ② SongsRepository lazily injected to mitigate the recursive dependency between the Impl class and the full repository instance

322.8. Calling Custom Query

With all that in place, we can then call our custom random() method and obtain a sample Song to work with from the database.

Example Custom Query Client Call

```
//when
Optional<Song> randomSong = songsRepository.random();
//then
then(randomSong.isPresent()).isTrue();
```

The following shows the resulting SQL

Custom Random Query Resulting SQL

```
select count(*) from reposongs_song s1_0
select ...
from reposongs_song s1_0
offset ? rows fetch first ? rows only
```

Chapter 323. Summary

In this module, we learned:

- that Spring Data JPA eliminates the need to write boilerplate JPA code
- to perform basic CRUD management for `@Entity` classes using a repository
- to implement query-by-example
- unbounded collections can grow over time and cause our applications to eventually fail
 - that paging and sorting can easily be used with repositories
- to implement query methods derived from a query DSL
- to implement custom repository extensions

323.1. Comparing Query Types

Of the query types,

- derived queries and query-by-example are simpler but have their limits
 - derived queries are more expressive
 - query-by-example can be built flexibly at runtime
 - nothing is free—so anything that requires translation between source and JPA form may incur extra initialization and/or processing time
- JPA-QL and native SQL
 - have virtually no limit to what they can express
 - cannot be dynamically defined for a repository like query-by-example. You would need to use the `EntityManager` directly to do that.
 - have loose coupling between the repository method name and the actual function of the executed query
 - can be resolved in an external source file that would allow for query changes without recompiling

JPA Repository End-to-End Application

copyright © 2024 jim stafford (jim.stafford@jhu.edu)

Chapter 324. Introduction

This lecture takes what you have learned in establishing a RDBMS data tier using Spring Data JPA and shows that integrated into an end-to-end application with API CRUD calls and finder calls using paging. It is assumed that you already know about API topics like Data Transfer Objects (DTOs), JSON and XML content, marshalling/unmarshalling using Jackson and JAXB, web APIs/controllers, and clients. This lecture will put them all together.

324.1. Goals

The student will learn:

- to integrate a Spring Data JPA Repository into an end-to-end application, accessed through an API
- to make a clear distinction between Data Transfer Objects (DTOs) and Business Objects (BOs)
- to identify data type architectural decisions required for a multi-tiered application
- to understand the need for paging when working with potentially unbounded collections and remote clients
- to setup proper transaction and other container feature boundaries using annotations and injection

324.2. Objectives

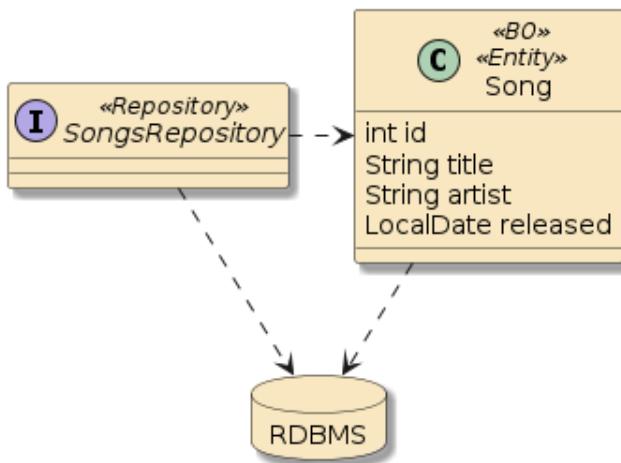
At the conclusion of this lecture and related exercises, the student will be able to:

1. implement a BO tier of classes that will be mapped to the database
2. implement a DTO tier of classes that will exchange state with external clients
3. implement a service tier that completes useful actions
4. identify the controller/service layer interface decisions when it comes to using DTO and BO classes
5. determine the correct transaction propagation property for a service tier method
6. implement a mapping tier between BO and DTO objects
7. implement paging requests through the API
8. implement page responses through the API

Chapter 325. BO/DTO Component Architecture

325.1. Business Object(s)/@Entities

For our Songs application—I have kept the data model simple and kept it limited to a single business object (BO) `@Entity` class mapped to the database using JPA and accessed through a Spring Data JPA repository.



The business objects are the focal point of information where we implement our business decisions.

Figure 137. BO Class Mapped to DB as JPA `@Entity`

The primary focus of our BO classes is to map business implementation concepts to the database. There are two fundamental patterns of business objects:

- **Anemic Domain Model** - containing no validations, calculations, or implementation of business rules. A basic data mapping with getters and setters.
- **Rich Domain Model** - combining data with business behavior indicative of an Object-Oriented design. The rich domain model is at the heart of Domain Driven Design (DDD) architectural concepts.

Due to our simplistic business domain, the example business object is very anemic. Do not treat that as a desirable target for all cases.

The following snippet shows some of the required properties of a JPA `@Entity` class.

BO Class Sample JPA Mappings

```
@Entity  
@Table(name="REPOSONGS_SONG")  
@NoArgsConstructor  
...  
@SequenceGenerator(name="REPOSONGS_SONG_SEQUENCE", allocationSize = 50)  
public class Song {  
    @Id @GeneratedValue(strategy = GenerationType.SEQUENCE,  
        generator = "REPOSONGS_SONG_SEQUENCE")
```

```

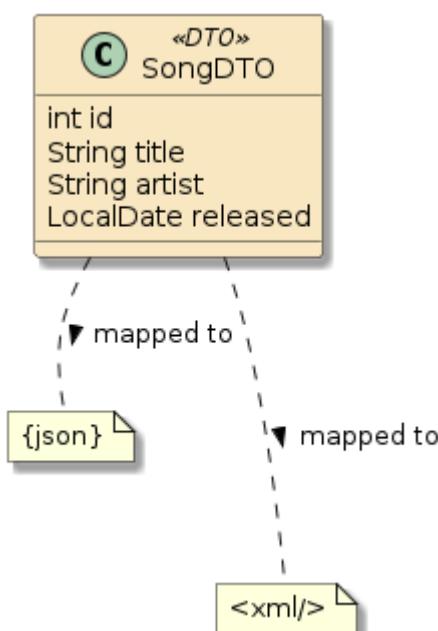
private int id;
...

```

325.2. Data Transfer Object(s) (DTOs)

The Data Transfer Objects are the focal point of interfacing with external clients. They represent state at a point in time. For external web APIs, they are commonly mapped to both JSON and XML.

For the API, we have the decision of whether to reuse BO classes as DTOs or implement a separate set of classes for that purpose. Even though some applications start out simple, there will come a point where database technology or mappings will need to change at a different pace than API technology or mappings.



For that reason, I created a separate SongsDTO class to represent a sample DTO. It has a near 1:1 mapping with the Song BO. This 1:1 representation of information makes it seem like this is an unnecessary extra class, but it demonstrates an initial technical separation between the DTO and BO that allows for independent changes down the road.

Figure 138. DTO

The primary focus of our DTO classes is to map business interface concepts to a portable exchange format.

The following snippet shows some of the annotations required to map the `SongDTO` class to XML using Jackson and JAXB. Jackson JSON requires very few annotations in the simple cases.

DTO Class Sample JSON/XML Mappings

```

import com.fasterxml.jackson.dataformat.xml.annotation.JacksonXmlRootElement;
import com.fasterxml.jackson.dataformat.xml.annotation.JacksonXmlProperty;
import jakarta.xml.bind.annotation.XmlRootElement;
import jakarta.xml.bind.annotation.XmlAccessType;
import jakarta.xml.bind.annotation.XmlAccessorType;
import jakarta.xml.bind.annotation.XmlAttribute;
...
@JacksonXmlRootElement(localName = "song", namespace = "urn:ejava.db-repo.songs")
@XmlRootElement(name = "song", namespace = "urn:ejava.db-repo.songs") ②
@NoArgsConstructor

```

```

...
public class SongDTO { ①
    @JacksonXmlProperty(isAttribute = true)
    @XmlAttribute
    private int id;
    private String title;
    private String artist;
    @XmlJavaTypeAdapter(LocalDateJaxbAdapter.class) ③
    private LocalDate released;
...

```

① Jackson JSON requires very little to no annotations for simple mappings

② XML mappings require more detailed definition to be complete

③ JAXB requires a custom mapping definition for java.time types

325.3. BO/DTO Mapping

With separate BO and DTO classes, there is a need for mapping between the two.

- map from DTO to BO for requests
- map from BO to DTO for responses

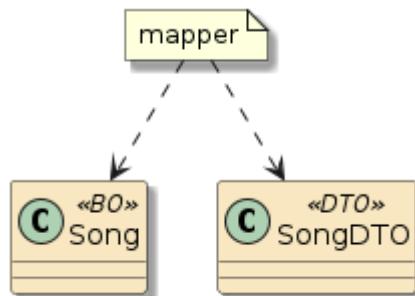


Figure 139. BO to DTO Mapping

We have several options on how to organize this role.

325.3.1. BO/DTO Self Mapping

- The BO or the DTO class can map to the other
 - Benefit: good encapsulation of detail within the data classes themselves
 - Drawback: promotes coupling between two layers we were trying to isolate



Avoid unless users of DTO will be tied to BO and are just exchanging information.

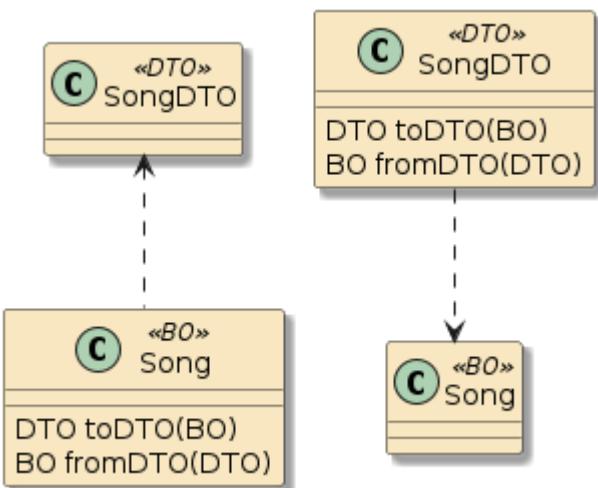


Figure 140. BO to DTO Self Mapping

325.3.2. BO/DTO Method Self Mapping

- The API or service methods can map things themselves within the body of the code
 - Benefit: mapping specialized to usecase involved
 - Drawback:
 - mixed concerns within methods.
 - likely have repeated mapping code in many methods



Avoid.

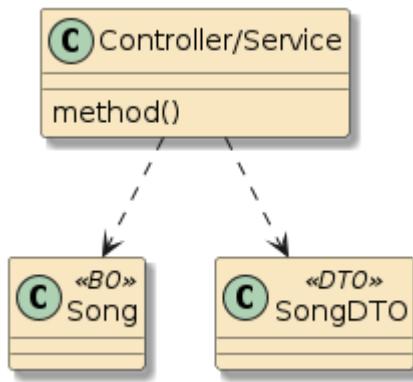


Figure 141. BO to DTO Method Self Mapping

325.3.3. BO/DTO Helper Method Mapping

- Delegate mapping to a reusable helper method within the API or service classes
 - Benefit: code reuse within the API or service class
 - Drawback: potential for repeated mapping in other classes



This is a small but significant step to a helper class

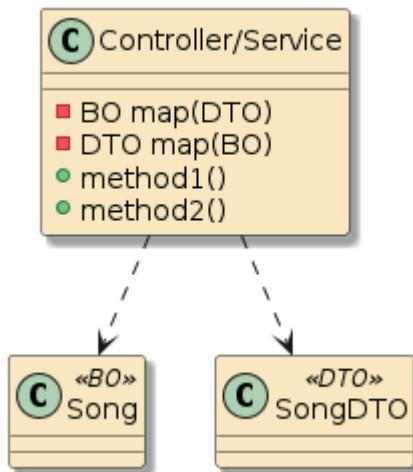


Figure 142. BO/DTO Helper Method Mapping

325.3.4. BO/DTO Helper Class Mapping

- Create a separate interface/class to inject into the API or service classes that encapsulates the role of mapping
 - Benefit: Reusable, testable, separation of concern
 - Drawback: none



Best in most cases unless good reason for self-mapping is appropriate.

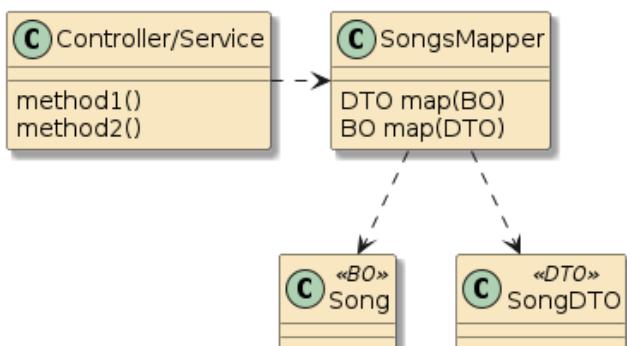


Figure 143. BO/DTO Helper Class Mapping

325.3.5. BO/DTO Helper Class Mapping Implementations

Mapping helper classes can be implemented by:

- brute force implementation
 - Benefit: likely the fastest performance and technically simplest to understand
 - Drawback: tedious setter/getter code
- off-the-shelf mapper libraries (e.g. [Dozer](#), [Orika](#), [MapStruct](#), [ModelMapper](#), [JMapper](#)) ^[1] ^[2]
 - Benefit:
 - declarative language and inferred DIY mapping options
 - some rely on code generation at compile time (similar in lifecycle to Lombok in some ways) with the ability to override and customize
 - Drawbacks:
 - some rely on reflection for mapping which add to overhead
 - non-trivial mappings can be complex to understand



MapStruct Thumbs Up

I have personally used Dozer in detail (years ago) and have recently been introduced to MapStruct. I really like MapStruct much better. It writes much of the same code you would have written in the brute force approach—without using reflection at runtime. You can define a mapper through interfaces and abstract classes—depending on how much you need to customize. You can also declare the mapper as a component to have helper components injected for use in mapping. In the end, you get a class with methods written in Java source that you can clearly see. Everything is understandable.

[1] "Performance of Java Mapping Frameworks", Baeldung

[2] "any tool for java object to object mapping?", Stack Overflow

Chapter 326. Service Architecture

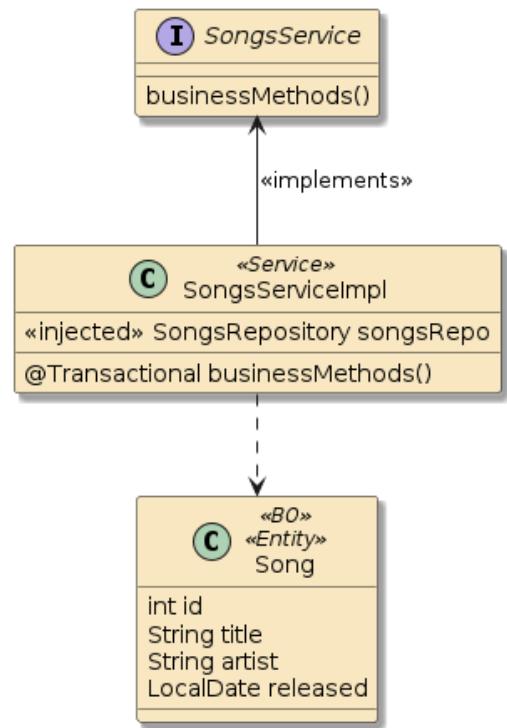
Services—with the aid of BOs—implement the meat of the business logic.

The service

- implements an interface with business methods
- is annotated with `@Service` component in most cases to self-support auto-injection (or use `@Bean` factory)
- injects repository component(s)
- declares transaction boundaries on methods
- interacts with BO instances

Example Service Class Declaration

```
@RequiredArgsConstructor  
@Service  
public class SongsServiceImpl  
    implements SongsService {  
    private final SongsMapper mapper;  
    private final SongsRepository songsRepo;  
    ...
```



326.1. Injected Service Boundaries

Container features like `@Transactional`, `@PreAuthorize`, `@Async`, etc. are only implemented at component boundaries. When a `@Component` dependency is injected, the container has the opportunity to add features using "*interpose*". As a part of interpose—the container implements proxy to add the desired feature of the target component method.

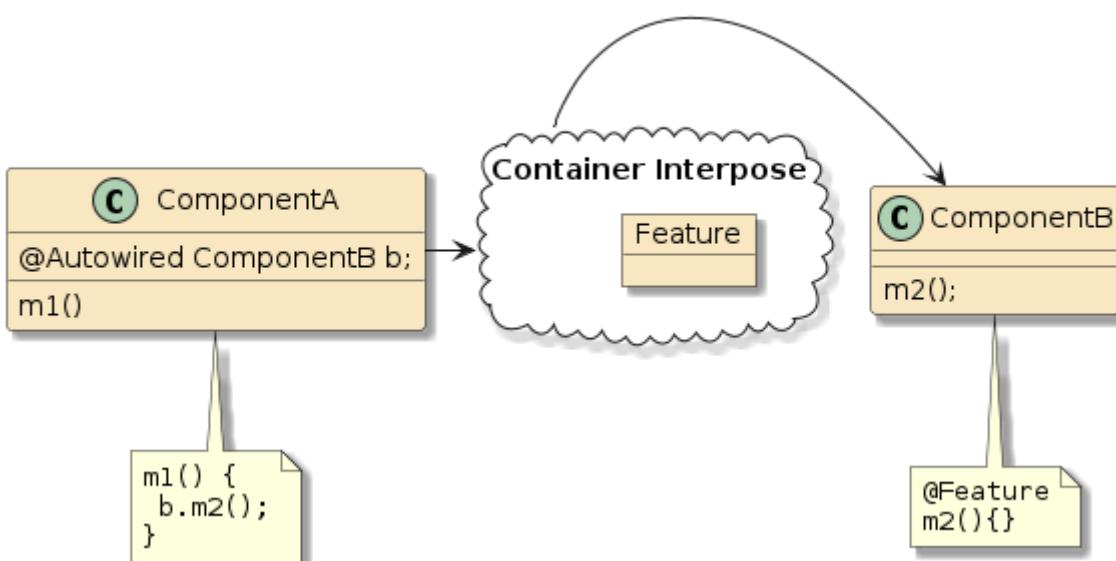


Figure 144. Container Interpose

Therefore it is important to arrange a component boundary wherever you need to start a new characteristic provided by the container. The following is a more detailed explanation of what not to do and do.

326.1.1. Buddy Method Boundary

The methods within a component class are not typically subject to container interpose. Therefore a call from m1() to m2() within the same component class is a straight Java call.



No Interpose for Buddy Method Calls

Buddy method calls are straight Java calls without container interpose.

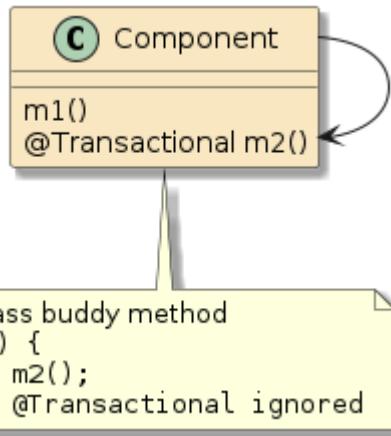


Figure 145. Buddy Method Boundary

326.1.2. Self Instantiated Method Boundary

Container interpose is only performed when the container has a chance to decorate the called component. Therefore, a call to a method of a component class that is self-instantiated will not have container interpose applied—no matter how the called method is annotated.



No Interpose for Self-Instantiated Components

Self-instantiated classes are not subject to container interpose.

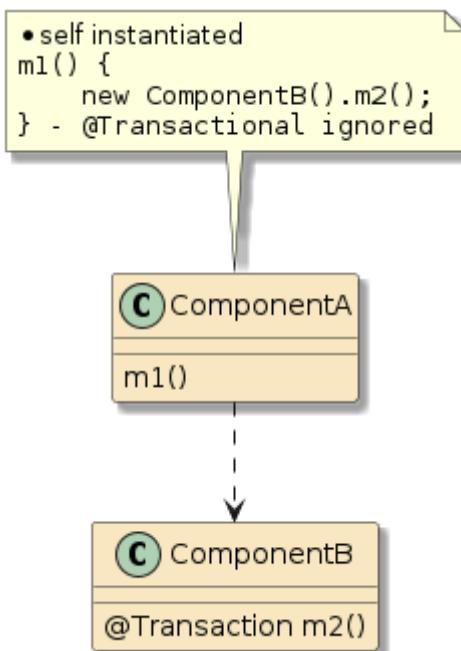


Figure 146. Self Instantiated Method Boundary

326.1.3. Container Injected Method Boundary

Components injected by the container are subject to container interpose and will have declared characteristics applied.



*Container-Injected Components
have Interpose*

Use container injection to have declared features applied to called component methods.

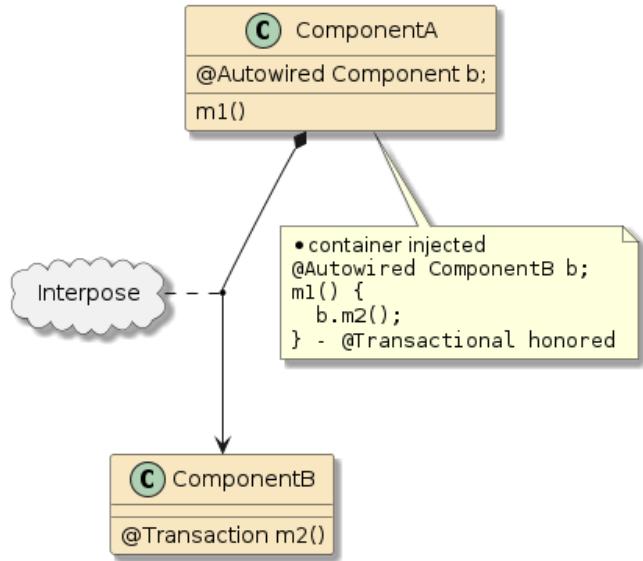


Figure 147. Container Injected Method Boundary

326.2. Compound Services

With `@Component` boundaries and interpose constraints understood—in more complex transaction, security, or threading solutions, the **logical** `@Service` many get broken up into one or more **physical** helper `@Component` classes.

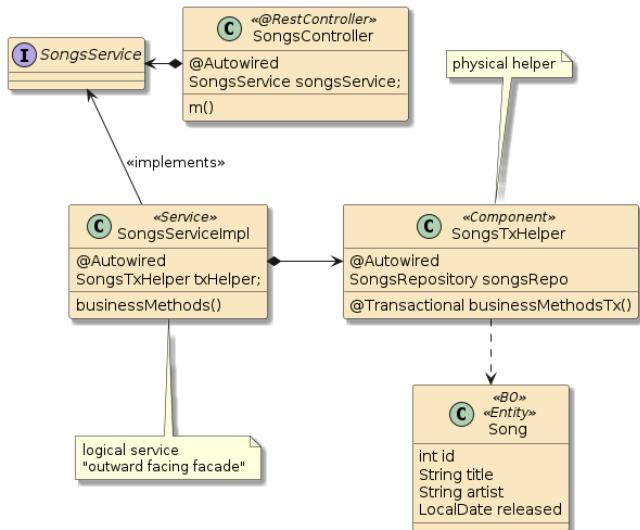


Figure 148. Single Service Expressed as Multiple Components

Each physical helper `@Component` is primarily designed around container augmentation (ex. action(s) to be performed within a single `@Transaction`). The remaining parts of the logical service are geared towards implementing the outward facing facade, and integrating the methods of the helper(s) to complete the intended role of the service. An example of this would be large loops of behavior.

```
for (...) { txHelper.txMethod(); }
```

To external users of `@Service`—it is still logically, just one `@Service`.

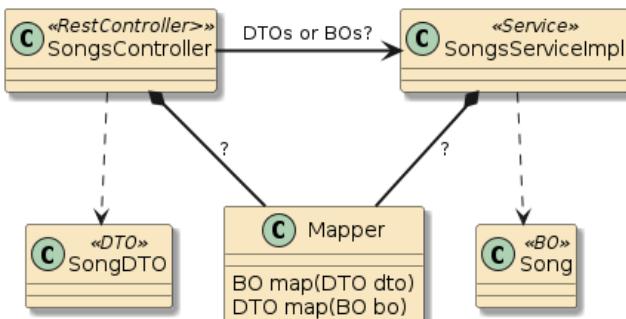
Conceptual Services may be broken into Multiple Physical Components

Conceptual boundaries for a service usually map 1:1 with a single physical class. However, there are cases when the conceptual service needs to be implemented by multiple physical classes/**@Components**.



Chapter 327. BO/DTO Interface Options

With the core roles of BOs and DTOs understood, we next have a decision to make about where to use them within our application between the API and service classes.



- Controller external interface will always be based on DTOs.
- Service's internal implementation will always be based on BOs.
- Where do we make the transition?

Figure 149. BO/DTO Interface Decisions

327.1. API Maps DTO/BO

It is natural to think of the `@Service` as working with pure implementation (BO) classes. This leaves the mapping job to the `@RestController` and all clients of the `@Service`.

- Benefit: If we wire two `@Services` together, they could efficiently share the same BO instances between them with no translation.
- Drawback: `@Services` should be the boundary of a solution and encapsulate the implementation details. BOs leak implementation details.

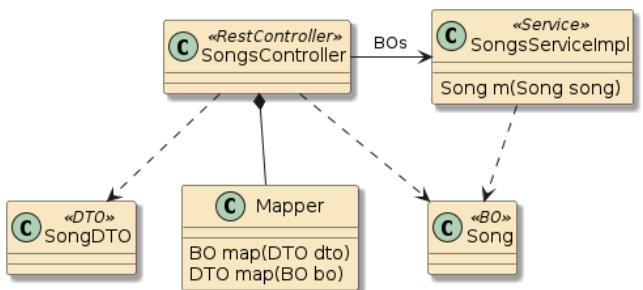
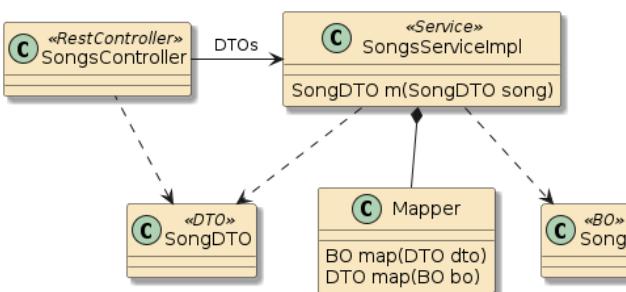


Figure 150. API Maps DTO to BO for Service Interface

327.2. @Service Maps DTO/BO

Alternatively, we can have the `@Service` fully encapsulate the implementation details and work with DTOs in its interface. This places the job of DTO/BO translation to the `@Service` and the `@RestController` and all `@Service` clients work with DTOs.

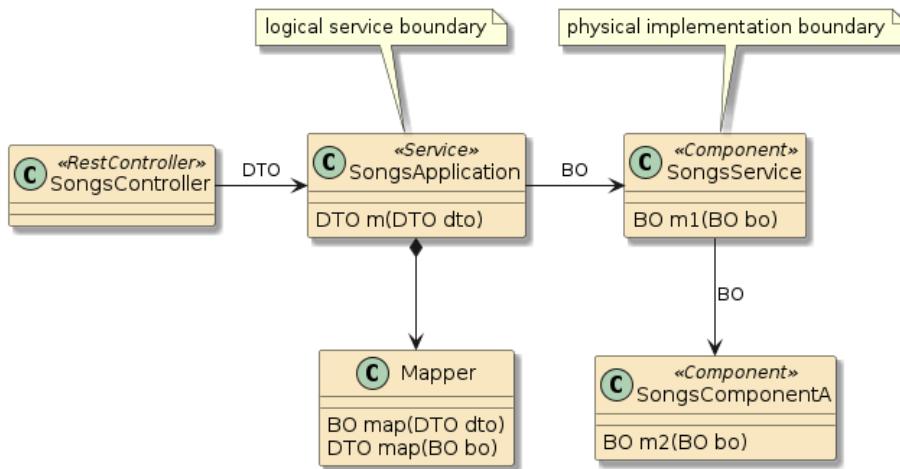


- Benefit: `@Service` fully encapsulates implementation and exchanges information using DTOs designed for interfaces.
- Drawback: BOs go through a translation when passing from `@Service` to `@Service` directly.

Figure 151. Service Maps DTO in Service Interface to BO

327.3. Layered Service Mapping Approach

The later DTO interface/mapping approach just introduced—maps closely to the [Domain Driven Design \(DDD\)](#) "Application Layer". However, one could also implement a layering of services.



- outer **@Service** classes represent the boundary to the application and interface using DTOs
- inner **@Component** classes represent implementation components and interface using native BOs

Layered Services Permit a Level of Trust between Inner Components

When using this approach, I like:



- all normalization and validation complete by the time DTOs are converted to BOs in the Application tier
- BOs exchanged between implementation components assume values are valid and normalized

Chapter 328. Implementation Details

With architectural decisions understood, lets take a look at some of the key details of the end-to-end application.

328.1. Song BO

We have already covered the `Song` BO `@Entity` class in a lot of detail during the JDBC, JPA, and Spring Data JPA lectures. The following lists most of the key business aspects and implementation details of the class.

Song BO Class with JPA Database Mappings

```
package info.ejava.examples.db.repo.jpa.songs.bo;  
...  
@Entity  
@Table(name="REPOSONGS_SONG")  
@Getter  
@ToString  
@Builder  
@With  
@AllArgsConstructor  
@NoArgsConstructor  
...  
@SequenceGenerator(name="REPOSONGS_SONG_SEQUENCE", allocationSize = 50)  
public class Song {  
    @Id @GeneratedValue(strategy = GenerationType.SEQUENCE,  
                        generator = "REPOSONGS_SONG_SEQUENCE")  
    @Column(name="ID", nullable=false, insertable=true, updatable=false)  
    private int id;  
    @Setter  
    @Column(name="TITLE", length=255, nullable=true, insertable=true, updatable=true)  
    private String title;  
    @Setter  
    private String artist;  
    @Setter  
    private LocalDate released;  
}
```

328.2. SongDTO

The `SongDTO` class has been mapped to Jackson JSON and Jackson and JAXB XML. The details of Jackson and JAXB mapping were covered in the API Content lectures. Jackson JSON required no special annotations to map this class. Jackson and JAXB XML primarily needed some annotations related to namespaces and attribute mapping. JAXB also required annotations for mapping the `LocalDate` field.

The following lists the annotations required to marshal/unmarshal the `SongsDTO` class using

Jackson and JAXB.

SongDTO Class with JSON and XML Mappings

```
package info.ejava.examples.db.repo.jpa.songs.dto;  
...  
@JacksonXmlRootElement(localName = "song", namespace = "urn:ejava.db-repo.songs")  
@XmlRootElement(name = "song", namespace = "urn:ejava.db-repo.songs")  
@XmlAccessorType(XmlAccessType.FIELD)  
@Data @Builder  
@NoArgsConstructor @AllArgsConstructor  
public class SongDTO {  
    @JacksonXmlProperty(isAttribute = true)  
    @XmlAttribute  
    private int id;  
    private String title;  
    private String artist;  
    @XmlJavaTypeAdapter(LocalDateJaxbAdapter.class) ①  
    private LocalDate released;  
...  
}
```

① JAXB requires an adapter for the newer LocalDate java class

328.2.1. LocalDateJaxbAdapter

Jackson is configured to marshal LocalDate out of the box using the ISO_LOCAL_DATE format for both JSON and XML.

ISO_LOCAL_DATE format

```
"released" : "2013-01-30"      //Jackson JSON  
<released xmlns="">2013-01-30</released> //Jackson XML
```

JAXB does not have a default format and requires the class be mapped to/from a string using an **XmlAdapter**.

LocalDateJaxbAdapter Class

```
@XmlJavaTypeAdapter(LocalDateJaxbAdapter.class) ①  
private LocalDate released;  
  
public static class LocalDateJaxbAdapter extends XmlAdapter<String, LocalDate> {②  
    @Override  
    public LocalDate unmarshal(String text) {  
        return null!=text ? LocalDate.parse(text, DateTimeFormatter.ISO_LOCAL_DATE) :  
null;  
    }  
    @Override  
    public String marshal(LocalDate timestamp) {
```

```

        return null!=timestamp ? DateTimeFormatter.ISO_LOCAL_DATE.format(timestamp) :
    null;
    }
}

```

- ① JAXB requires an adapter to translate from `LocalDate` to/from XML
- ② we can define an `XmlAdapter` to address `LocalDate` using `java.time` classes

328.3. Song JSON Rendering

The following snippet provides example JSON of a `Song` DTO payload.

Song JSON Rendering

```
{
    "id" : 1,
    "title" : "Tender Is the Night",
    "artist" : "No Doubt",
    "released" : "2003-11-16"
}
```

328.4. Song XML Rendering

The following snippets provide example XML of `Song` DTO payloads. They are technically equivalent from an XML Schema standpoint, but use some alternate syntax XML to achieve the same technical goals.

Song Jackson XML Rendering

```
<song xmlns="urn:ejava.db-repo.songs" id="2">
    <title>The Mirror Crack'd from Side to Side</title>
    <artist>Earth Wind and Fire</artist>
    <released>2018-01-01</released>
</song>
```

Song JAXB XML Rendering

```
<ns2:song xmlns:ns2="urn:ejava.db-repo.songs" id="1">
    <title>Brandy of the Damned</title>
    <artist>Orbital</artist>
    <released>2015-11-10</released>
</ns2:song>
```

328.5. Pageable/PageableDTO

I placed a high value on paging when working with unbounded collections when covering

repository find methods. The value of paging comes especially into play when dealing with external users. That means we will need a way to represent Page, Pageable, and Sort in requests and responses as a part of DTO solution.

You will notice that I made a few decisions on how to implement this interface

1. I am assuming that both sides of the interface using the DTO classes are using Spring Data.
2. I am using the Page, Pageable, and Sort DTOs to directly self-map to/from Spring Data types. This makes the client and service code much simpler.

```
Pageable pageable = PageableDTO.of(pageNumber, pageSize, sortString).toPageable();  
①  
Page<SongDTO> result = ...  
SongsPageDTO resultDTO = new SongsPageDTO(result); ①
```

① using self-mapping between paging DTOs and Spring Data (Pageable and Page) types

3. I chose to use the Spring Data types (Pageable and Page) in the @Service interface when expressing paging and performed the Spring Data/DTO mappings in the @RestController. The @Service still takes DTO business types and maps DTO business types to/from BOs. I did this so that I did not eliminate any pre-existing library integration with Spring Data paging types.

```
Page<SongDTO> getSongs(Pageable pageable); ①
```

① using Spring Data (Pageable and Page) and business DTO (SongDTO) types in @Service interface

I will be going through the architecture and wiring in these lecture notes. The actual DTO code is surprisingly complex to render in the different formats and libraries. These topics were covered in detail in the API content lectures. I also chose to implement the PageableDTO and sort as immutable—which added some interesting mapping challenges worth inspecting.

328.5.1. PageableDTO Request

Requests require an expression for Pageable. The most straight forward way to accomplish this is through query parameters. The example snippet below shows pageNumber, pageSize, and sort expressed as simple string values as part of the URI. We have to write code to express and parse that data.

Example Pageable Query Parameters

```
①  
/api/songs/example?pageNumber=0&pageSize=5&sort=released:DESC,id:ASC  
②
```

① pageNumber and pageSize are direct properties used by PageRequest

② sort contains a comma separated list of order compressed into a single string

Integer pageNumber and pageSize are straight forward to represent as numeric values in the query.

Sort requires a minor amount of work. Spring Data Sort is an ordered list of "property and direction". I have chosen to express property and direction using a ":" separated string and concatenate the ordering using a ",". This allows the query string to be expressed in the URI without special characters.

328.5.2. PageableDTO Client-side Request Mapping

Since I expect code using the PageableDTO to also be using Spring Data, I chose to use self-mapping between the PageableDTO and Spring Data Pageable.

The following snippet shows how to map **Pageable** to PageableDTO and the PageableDTO properties to URI query parameters.

Building URI with Pageable Request Parameters

```
PageRequest pageable = PageRequest.of(0, 5,
    Sort.by(Sort.Order.desc("released"), Sort.Order.asc("id")));
PageableDTO pageSpec = PageableDTO.of(pageable); ①
URI uri=UriComponentsBuilder
    .fromUri(serverConfig.getBaseUrl())
    .path(SongsController.SONGS_PATH).path("/example")
    .queryParams(pageSpec.getQueryParams()) ②
    .build().toUri();
```

① using PageableDTO to self map from Pageable

② using PageableDTO to self map to URI query parameters

328.5.3. PageableDTO Server-side Request Mapping

The following snippet shows how the individual page request properties can be used to build a local instance of PageableDTO in the **@RestController**. Once the PageableDTO is built, we can use that to self map to a Spring Data **Pageable** to be used when calling the **@Service**.

```
public ResponseEntity<SongsPageDTO> findSongsByExample(
    @RequestParam(value="pageNumber",defaultValue="0",required=false) Integer
pageNumber,
    @RequestParam(value="pageSize",required=false) Integer pageSize,
    @RequestParam(value="sort",required=false) String sortString,
    @RequestBody SongDTO probe) {

    Pageable pageable = PageableDTO.of(pageNumber, pageSize, sortString) ①
        .toPageable(); ②
```

① building PageableDTO from page request properties

② using PageableDTO to self map to Spring Data Pageable

328.5.4. Pageable Response

Responses require an expression for Pageable to indicate the pageable properties about the content returned. This must be expressed in the payload, so we need a JSON and XML expression for this. The snippets below show the JSON and XML DTO renderings of our Pageable properties.

Example JSON Pageable Response Document

```
"pageable" : {  
    "pageNumber" : 1,  
    "pageSize" : 25,  
    "sort" : "title:ASC,artist:ASC"  
}
```

Example XML Pageable Response Document

```
<pageable xmlns="urn:ejava.common.dto" pageNumber="1" pageSize="25" sort=  
"title:ASC,artist:ASC"/>
```

328.6. Page/PageDTO

Pageable is part of the overall Page<T>, with contents. Therefore, we also need a way to return a page of content to the caller.

328.6.1. PageDTO Rendering

JSON is very lenient and could have been implemented with a generic PageDTO<T> class.

```
{"content": [ ①  
    {"id": 10, ②  
        "title": "Blue Remembered Earth",  
        "artist": "Coldplay",  
        "released": "2009-03-18"}],  
    "totalElements": 10, ①  
    "pageable": {"pageNumber": 3, "pageSize": 3, "sort": null} ①  
}
```

① content, totalElements, and pageable are part of reusable PageDTO

② song within content array is part of concrete Songs domain

However, XML—with its use of unique namespaces, requires a sub-class to provide the type-specific values for content and overall page.

```
<songsPage xmlns="urn:ejava.db-repo.songs" totalElements="10"> ①  
    <wstxns1:content xmlns:wstxns1="urn:ejava.common.dto">  
        <song id="10"> ②  
            <title xmlns="">Blue Remembered Earth</title>
```

```

<artist xmlns="">Coldplay</artist>
<released xmlns="">2009-03-18</released>
</song>
</wstxns1:content>
<pageable xmlns="urn:ejava.common.dto" pageNumber="3" pageSize="3"/>
</songsPage>

```

- ① `totalElements` mapped to XML as an (optional) attribute
- ② `songsPage` and `song` are in concrete domain `urn:ejava.db-repo.songs` namespace

328.6.2. SongsPageDTO Subclass Mapping

The `SongsPageDTO` subclass provides the type-specific mapping for the content and overall page. The generic portions are handled by the base class.

SongsPageDTO Subclass Mapping

```

@JacksonXmlElementWrapper(localName = "songsPage", namespace = "urn:ejava.db-repo.songs")
①
@XmlRootElement(name = "songsPage", namespace = "urn:ejava.db-repo.songs") ①
@XmlType(name = "SongsPage", namespace = "urn:ejava.db-repo.songs")
@XmlAccessorType(XmlAccessType.NONE)
@NoArgsConstructor
public class SongsPageDTO extends PageDTO<SongDTO> {
    @JsonProperty
    @JacksonXmlElementWrapper(localName = "content", namespace = "
urn:ejava.common.dto")②
        @JacksonXmlProperty(localName = "song", namespace = "urn:ejava.db-repo.songs") ③
        @XmlElementWrapper(name="content", namespace = "urn:ejava.common.dto") ②
        @XmlElement(name="song", namespace = "urn:ejava.db-repo.songs") ③
    public List<SongDTO> getContent() {
        return super.getContent();
    }
    public SongsPageDTO(List<SongDTO> content, Long totalElements, PageableDTO
pageableDTO) {
        super(content, totalElements, pageableDTO);
    }
    public SongsPageDTO(Page<SongDTO> page) {
        this(page.getContent(), page.getTotalElements(),
            PageableDTO.fromPageable(page.getPageable()));
    }
}

```

- ① Each type-specific mapping must have its own XML naming
- ② "Wrapper" is the outer element for the individual members of collection and part of generic framework
- ③ "Property/Element" is the individual members of collection and interface/type specific

328.6.3. PageDTO Server-side Rendering Response Mapping

The `@RestController` can use the concrete DTO class (`SongPageDTO` in this case) to self-map from a Spring Data `Page<T>` to a DTO suitable for marshaling back to the API client.

PageDTO Server-side Response Mapping

```
Page<SongDTO> result=songsService.findSongsMatchingAll(probe, pageable);
SongsPageDTO resultDTO = new SongsPageDTO(result); ①
ResponseEntity<SongsPageDTO> response = ResponseEntity.ok(resultDTO);
```

① using `SongsPageDTO` to self-map Sing Data `Page<T>` to DTO

328.6.4. PageDTO Client-side Rendering Response Mapping

The `PageDTO<T>` class can be used to self-map to a Spring Data `Page<T>`. `Pageable`, if needed, can be obtained from the `Page<T>` or through the `pageDTO.getPageable()` DTO result.

PageDTO Client-side Response Mapping

```
//when
SongsPageDTO pageDTO = restTemplate.exchange(request, SongsPageDTO.class).getBody();
//then
Page<SongDTO> page = pageDTO.toPage(); ①
then(page.getSize()).isEqualTo(pageableRequest.getPageSize());
then(page.getNumber()).isEqualTo(pageableRequest.getPageNumber());
then(page.getSort()).isEqualTo(Sort.by(Sort.Direction.DESC, "released"));
Pageable pageable = page.getPageable(); ②
```

① using `PageDTO<T>` to self-map to a Spring Data `Page<T>`

② can use `page.getPageable()` or `pageDTO.getPageable().toPageable()` obtain `Pageable`

Chapter 329. SongMapper

The `SongMapper` `@Component` class is used to map between `SongDTO` and `Song` BO instances. It leverages Lombok builder methods — but is pretty much a simple/brute force mapping.

329.1. Example Map: SongDTO to Song BO

The following snippet is an example of mapping a `SongDTO` to a `Song` BO.

Map SongDTO to Song BO

```
@Component
public class SongsMapper {
    public Song map(SongDTO dto) {
        Song bo = null;
        if (dto!=null) {
            bo = Song.builder()
                .id(dto.getId())
                .artist(dto.getArtist())
                .title(dto.getTitle())
                .released(dto.getReleased())
                .build();
        }
        return bo;
    }
    ...
}
```

329.2. Example Map: Song BO to SongDTO

The following snippet is an example of mapping a `Song` BO to a `SongDTO`.

Map Song BO to SongDTO

```
...
public SongDTO map(Song bo) {
    SongDTO dto = null;
    if (bo!=null) {
        dto = SongDTO.builder()
            .id(bo.getId())
            .artist(bo.getArtist())
            .title(bo.getTitle())
            .released(bo.getReleased())
            .build();
    }
    return dto;
}
...
```

Chapter 330. Service Tier

The SongsService `@Service` encapsulates the implementation of our management of Songs.

330.1. SongsService Interface

The `SongsService` interface defines a portion of pure CRUD methods and a series of finder methods. To be consistent with DDD encapsulation, the `@Service` interface is using DTO classes. Since the `@Service` is an injectable component, I chose to use straight Spring Data pageable types to possibly integrate with libraries that inherently work with Spring Data types.

SongsService Interface

```
public interface SongsService {  
    SongDTO createSong(SongDTO songDTO); ①  
    SongDTO getSong(int id);  
    void updateSong(int id, SongDTO songDTO);  
    void deleteSong(int id);  
    void deleteAllSongs();  
  
    Page<SongDTO> findReleasedAfter(LocalDate exclusive, Pageable pageable);②  
    Page<SongDTO> findSongsMatchingAll(SongDTO probe, Pageable pageable);  
}
```

① chose to use DTOs for business data (`SongDTO`) in `@Service` interface

② chose to use Spring Data types (`Page` and `Pageable`) in pageable `@Service` finder methods

330.2. SongsServiceImpl Class

The `SongsServiceImpl` implementation class is implemented using the `SongsRepository` and `SongsMapper`.

SongsServiceImpl Implementation Attributes

```
@RequiredArgsConstructor ① ②  
@Service  
public class SongsServiceImpl implements SongsService {  
    private final SongsMapper mapper;  
    private final SongsRepository songsRepo;
```

① Creates a constructor for all final attributes

② Single constructors are automatically used for Autowiring

I will demonstrate two types of methods here — one requiring an active transaction and the other that only supports but does not require a transaction.

330.3. createSong()

The `createSong()` method

- accepts a `SongDTO`, creates a new song, and returns the created song as a `SongDTO`, with the generated ID.
- declares a `@Transaction` annotation to be associated with a Persistence Context and propagation `REQUIRED` in order to enforce that a database transaction be active from this point forward.
- calls the mapper to map from/to a `SongsDTO` to/from a `Song` BO
- uses the `SongsRepository` to interact with the database

`SongsServiceImpl.createSong()`

```
@Transactional(propagation = Propagation.REQUIRED) ① ② ③
public SongDTO createSong(SongDTO songDTO) {
    Song songBO = mapper.map(songDTO); ④

    //manage instance
    songsRepo.save(songBO); ⑤

    return mapper.map(songBO); ⑥
}
```

- ① `@Transaction` associates Persistence Context with thread of call
- ② `propagation` used to control activation and scope of transaction
- ③ `REQUIRED` triggers the transaction to start no later than this method
- ④ mapper converting DTO input argument to BO instance
- ⑤ BO instance saved to database and updated with primary key
- ⑥ mapper converting BO entity to DTO instance for return from service

330.4. findSongsMatchingAll()

The `findSongsMatchingAll()` method

- accepts a `SongDTO` as a probe and `Pageable` to adjust the search and results
- declares a `@Transaction` annotation to be associated with a Persistence Context and propagation `SUPPORTS` to indicate that no database changes will be performed by this method.
- calls the mapper to map from/to a `SongsDTO` to/from a `Song` BO
- uses the `SongsRepository` to interact with the database

`SongsServiceImpl Finder Method`

```
@Transactional(propagation = Propagation.SUPPORTS) ① ② ③
public Page<SongDTO> findSongsMatchingAll(SongDTO probeDTO, Pageable pageable) {
    Song probe = mapper.map(probeDTO); ④
```

```
ExampleMatcher matcher = ExampleMatcher.matchingAll().withIgnorePaths("id"); ⑤
Page<Song> songs = songsRepo.findAll(Example.of(probe, matcher), pageable); ⑥
return mapper.map(songs); ⑦
}
```

- ① `@Transaction` associates Persistence Context with thread of call
- ② `propagation` used to control activation and scope of transaction
- ③ `SUPPORTS` triggers the any active transaction to be inherited by this method but does not proactively start one
- ④ mapper converting DTO input argument to BO instance to create probe for match
- ⑤ building matching rules to include an ignore of `id` property
- ⑥ finder method invoked with matching and paging arguments to return page of BOs
- ⑦ mapper converting page of BOs to page of DTOs

Chapter 331. RestController API

The `@RestController` provides an HTTP Facade for our `@Service`.

`@RestController Class`

```
@RestController  
@Slf4j  
@RequiredArgsConstructor  
public class SongsController {  
    public static final String SONGS_PATH="api/songs";  
    public static final String SONG_PATH= SONGS_PATH + "/{id}";  
    public static final String RANDOM_SONG_PATH= SONGS_PATH + "/random";  
  
    private final SongsService songsService; ①
```

① `@Service` injected into class using constructor injection

I will demonstrate two of the operations available.

331.1. createSong()

The `createSong()` operation

- is called using `POST /api/songs` method and URI
- passed a SongDTO, containing the fields to use marshaled in JSON or XML
- calls the `@Service` to handle the details of creating the Song
- returns the created song using a SongDTO

`createSong()` API Operation

```
@RequestMapping(path=SONGS_PATH,  
    method=RequestMethod.POST,  
    consumes={MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE},  
    produces={MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE})  
public ResponseEntity<SongDTO> createSong(@RequestBody SongDTO songDTO) {  
  
    SongDTO result = songsService.createSong(songDTO); ①  
  
    URI uri = ServletUriComponentsBuilder.fromCurrentRequestUri()  
        .replacePath(SONG_PATH)  
        .build(result.getId()); ②  
    ResponseEntity<SongDTO> response = ResponseEntity.created(uri).body(result);  
    return response; ③  
}
```

① DTO from HTTP Request supplied to and result DTO returned from `@Service` method

② URI of created instance calculated for `Location` response header

③ DTO marshalled back to caller with HTTP Response

331.2. `findSongsByExample()`

The `findSongsByExample()` operation

- is called using "POST /api/songs/example" method and URI
- passed a `SongDTO` containing the properties to search for using JSON or XML
- calls the `@Service` to handle the details of finding the songs after mapping the `Pageable` from query parameters
- converts the `Page<SongDTO>` into a `SongsPageDTO` to address marshaling concerns relative to XML
- returns the page as a `SongsPageDTO`

findSongsByExample API Operation

```
@RequestMapping(path=SONGS_PATH + "/example",
    method=RequestMethod.POST,
    consumes={MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE},
    produces={MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE})
public ResponseEntity<SongsPageDTO> findSongsByExample(
    @RequestParam(value="pageNumber",defaultValue="0",required=false) Integer
pageNumber,
    @RequestParam(value="pageSize",required=false) Integer pageSize,
    @RequestParam(value="sort",required=false) String sortString,
    @RequestBody SongDTO probe) {

    Pageable pageable=PageableDTO.of(pageNumber, pageSize, sortString).toPageable();①
    Page<SongDTO> result=songsService.findSongsMatchingAll(probe, pageable); ②

    SongsPageDTO resultDTO = new SongsPageDTO(result); ③
    ResponseEntity<SongsPageDTO> response = ResponseEntity.ok(resultDTO);
    return response;
}
```

① `PageableDTO` constructed from page request query parameters

② `@Service` accepts DTO arguments for call and returns DTO constructs mixed with Spring Data paging types

③ type-specific `SongsPageDTO` marshalled back to caller to support type-specific XML namespaces

331.3. WebClient Example

The following snippet shows an example of using a `WebClient` to request a page of finder results from the API. `WebClient` is part of the Spring WebFlux libraries—which implements reactive streams. The use of `WebClient` here is purely for example and not a requirement of anything created. However, using `WebClient` did force my hand to add JAXB to the DTO mappings since

Jackson XML is not yet supported by WebFlux. RestTemplate does support both Jackson and JAXB XML mapping - which would have made mapping simpler.

WebClient Client

```
@Autowired
private WebClient webClient;
...
UriComponentsBuilder findByExampleUriBuilder = UriComponentsBuilder
    .fromUri(serverConfig.getBaseUrl())
    .path(SongsController.SONGS_PATH).path("/example");
...
//given
MediaType mediaType = ...
PageRequest pageable = PageRequest.of(0, 5, Sort.by(Sort.Order.desc("released")));
PageableDTO pageSpec = PageableDTO.of(pageable); ①
SongDTO allSongsProbe = SongDTO.builder().build(); ②
URI uri = findByExampleUriBuilder.queryParams(pageSpec.getQueryParams()) ③
    .build().toUri();
WebClient.RequestHeadersSpec<?> request = webClient.post()
    .uri(uri)
    .contentType(mediaType)
    .body(Mono.just(allSongsProbe), SongDTO.class)
    .accept(mediaType);
//when
ResponseEntity<SongsPageDTO> response = request
    .retrieve()
    .toEntity(SongsPageDTO.class).block();
//then
then(response.getStatusCode().is2xxSuccessful()).isTrue();
SongsPageDTO page = response.getBody();
```

① limiting query results to first page, ordered by "release", with a page size of 5

② create a "match everything" probe

③ pageable properties added as query parameters



WebClient/WebFlex does not yet support Jackson XML

WebClient and WebFlex does not yet support Jackson XML. This is what primarily forced the example to leverage JAXB for XML. WebClient/WebFlux automatically makes the decision/transition under the covers once an `@XmlElement` is provided.

Chapter 332. Summary

In this module, we learned:

- to integrate a Spring Data JPA Repository into an end-to-end application, accessed through an API
- implement a service tier that completes useful actions
- to make a clear distinction between DTOs and BOs
- to identify data type architectural decisions required for DTO and BO types
- to setup proper transaction and other container feature boundaries using annotations and injection
- implement paging requests through the API
- implement page responses through the API

MongoDB with Mongo Shell

copyright © 2024 jim stafford (jim.stafford@jhu.edu)

Chapter 333. Introduction

This lecture will introduce working with MongoDB database using the Mongo shell.

333.1. Goals

The student will learn:

- basic concepts behind the Mongo NoSQL database
- to create a database and collection
- to perform basic CRUD operations with database collection and documents using Mongo shell

333.2. Objectives

At the conclusion of this lecture and related exercises, the student will be able to:

1. identify the purpose of a MongoDB collection, structure of a MongoDB document, and types of example document fields
2. access a MongoDB database using the Mongo shell
3. perform basic CRUD actions on documents
4. perform paging commands
5. leverage the aggregation pipeline for more complex commands

Chapter 334. Mongo Concepts

Mongo is a document-oriented database. This type of database enforces very few rules when it comes to schema. About the only rules that exist are:

- a primary key field, called `_id` must exist
- no document can be larger than 16MB

GridFS API Supports Unlimited Size Documents.



MongoDB supports unlimited size documents using the [GridFS API](#). GridFS is basically a logical document abstraction over a collection of related individual physical documents — called "chunks" — abiding by the standard document-size limits

334.1. Mongo Terms

The table below lists a few keys terms associated with MongoDB.

Table 24. Mongo Terms

Mongo Term	Peer RDBMS	Description
	Term	
Database	Database	a group of document collections that fall under the same file and administrative management
Collection	Table	a set of documents with indexes and rules about how the documents are managed
Document	Row	a collection of fields stored in binary JSON (BSON) format. RDBMS tables must have a defined schema and all rows must match that schema.
Field	Column	a JSON property that can be a single value or nested document. An RDBMS column will have a single type based on the schema and cannot be nested.
Server	Server	a running instance that can perform actions on the database. A server contains more than one database.
Mongos	(varies)	an intermediate process used when data is spread over multiple servers. Since we will be using only a single server, we will not need a <code>mongos</code>

Mongo Term	Peer RDBMS	Description
	Term	
Database	Database	a group of document collections that fall under the same file and administrative management
Collection	Table	a set of documents with indexes and rules about how the documents are managed
Document	Row	a collection of fields stored in binary JSON (BSON) format. RDBMS tables must have a defined schema and all rows must match that schema.
Field	Column	a JSON property that can be a single value or nested document. An RDBMS column will have a single type based on the schema and cannot be nested.
Server	Server	a running instance that can perform actions on the database. A server contains more than one database.
Mongos	(varies)	an intermediate process used when data is spread over multiple servers. Since we will be using only a single server, we will not need a <code>mongos</code>

334.2. Mongo Documents

Mongo Documents are stored in a binary JSON format called "[BSON](#)". There are many [native types](#) that can be represented in BSON. Among them include "string", "boolean", "date", "ObjectId", "array", etc.

Documents/fields can be flat or nested.

Example Document

```
{  
    "field1": "value1", ①  
    "field2": "value2",  
    "field3": { ②  
        "field31": "value31",  
        "field32": "value32"  
    },  
    "field4": [ "value41", "value42", "value43" ], ③  
    "field5": [ ④  
        { "field511": "value511", "field512": "value512" },  
        { "field521": "value521" }  
        { "field531": "value531", "field532": "value532", "field533": "value533" }  
    ]  
}
```

① example field with value of BSON type

② example nested document within a field

③ example field with type array — with values of BSON type

④ example field with type array — with values of nested documents

The follow-on interaction examples will use a flat document structure to keep things simple to start with.

Chapter 335. MongoDB Server

To start our look into using Mongo commands, let's instantiate a MongoDB, connect with the Mongo Shell, and execute a few commands.

335.1. Starting Docker Compose MongoDB

One simple option we have to instantiate a MongoDB is to use Docker Compose.

The following snippet shows an example of launching MongoDB from the docker-compose.yml script in the example directory.

Starting Docker-Compose MongoDB

```
$ docker-compose up -d mongodb
Creating ejava_mongodb_1 ... done

$ docker ps --format "{{.Image}}\t{{.Ports}}\t{{.Names}}"
mongo:4.4.0-bionic 0.0.0.0:27017->27017/tcp mongo-book-example-mongodb-1 ①
```

① image is running with name `mongo-book-example-mongodb-1` and server `27017` port is mapped also to host

This specific MongoDB server is configured to use authentication and has an admin account pre-configured to use credentials `admin/secret`.

335.2. Connecting using Host's Mongo Shell

If we have Mongo shell installed locally, we can connect to MongoDB using the default mapping to localhost.

Connect using Host's Mongo shell

```
$ which mongo
/usr/local/bin/mongo ①
$ mongo -u admin -p secret ② ③
MongoDB shell version v4.4.0
connecting to:
mongodb://127.0.0.1:27017/?compressors=disabled&gssapiServiceName=mongodb
```

① mongo shell happens to be installed locally

② password can be securely prompted by leaving off command line

③ URL defaults to `mongodb://127.0.0.1:27017`

335.3. Connecting using Guest's Mongo Shell

If we do not have Mongo shell installed locally, we can connect to MongoDB by executing the

command in the MongoDB image.

Connecting using Guest's Mongo Shell

```
$ docker-compose exec mongodb mongo -u admin -p secret ① ②
MongoDB shell version v4.4.0
connecting to:
mongodb://127.0.0.1:27017/?compressors=disabled&gssapiServiceName=mongodb
```

① runs the `mongo` shell command within the `mongodb` Docker image

② URL defaults to `mongodb://127.0.0.1:27017`

335.4. Switch to test Database

We start off with three default databases meant primarily for server use.

Show Databases

```
> show dbs
admin   0.000GB
config   0.000GB
local   0.000GB
```

We can switch to a database to make it the default database for follow-on commands even before it exists.

Switch Database

```
> use test ①
switched to db test
> show collections
>
```

① makes the `test` database the default database for follow-on commands



Mongo will create a new/missing database on-demand when the first document is inserted.

335.5. Database Command Help

We can get a list of all commands available to us for a collection using the `db.<collection>.help()` command. The collection does not have to exist yet.

Get Collection Command Help

```
> db.books.help() ①
DBCollection help
...
```

```
db.books.insertOne( obj, <optional params> ) - insert a document, optional  
parameters are: w, wtimeout, j  
db.books.insert(obj)
```

① command to list all possible commands for a collection

Chapter 336. Basic CRUD Commands

336.1. Insert Document

We can create a new document in the database, stored in a named collection.

The following snippet shows the syntax for inserting a single, new book in the `books` collection. All fields are optional at this point and the `_id` field will be automatically generated by the server when we do not provide one.

Insert One Document

```
> db.books.insertOne({title:"GWW", author:"MM", published:ISODate("1936-06-30")})  
{  
  "acknowledged" : true,  
  "insertedId" : ObjectId("606c82da9ef76345a2bf0b7f") ①  
}
```

① `insertOne` command returns the `_id` assigned

MongoDB creates the collection, if it does not exist.

Created Collection

```
> show collections  
books
```

336.2. Primary Keys

MongoDB requires that all documents contain a primary key with the name `_id` and will generate one of type `ObjectId` if not provided. You have the option of using a business value from the document or a self-generated uniqueID, but it has to be stored in the `_id` field.

The following snippet shows an example of an `insert` using a supplied, numeric primary key.

Example Insert with Provided Primary Key

```
> db.books.insert({_id:17, title:"GWW", author:"MM", published:ISODate("1936-06-30")})  
WriteResult({ "nInserted" : 1 })  
  
> db.books.find({_id:17})  
{ "_id" : 17, "title" : "GWW", "author" : "MM", "published" : ISODate("1936-06-30T00:00:00Z") }
```

336.3. Document Index

All collections are required to have an index on the `_id` field. This index is generated automatically.

Default _id Index

```
> db.books.getIndexes()
[  
  { "v" : 2, "key" : { "_id" : 1 }, "name" : "_id_" } ①  
]
```

① index on `_id` field in `books` collection

336.4. Create Index

We can create an index on one or more other fields using the `createIndex()` command.

The following example creates a non-unique, ascending index on the `title` field. By making it sparse—only documents with a `title` field are included in the index.

Create Example Index

```
> db.books.createIndex({title:1}, {unique:false, sparse:true})
{
  "createdCollectionAutomatically" : false,
  "numIndexesBefore" : 1,
  "numIndexesAfter" : 2,
  "ok" : 1
}
```

336.5. Find All Documents

We can find all documents by passing in a JSON document that matches the fields we are looking for. We can find all documents in the collection by passing in an empty query (`{}`). Output can be made more readable by adding `.pretty()`.

Final All Documents

```
> db.books.find({}) ①
{ "_id" : ObjectId("606c82da9ef76345a2bf0b7f"), "title" : "GWW", "author" : "MM",
  "published" : ISODate("1936-06-30T00:00:00Z") }

> db.books.find({}).pretty() ②
{
  "_id" : ObjectId("606c82da9ef76345a2bf0b7f"),
  "title" : "GWW",
  "author" : "MM",
  "published" : ISODate("1936-06-30T00:00:00Z")
}
```

① empty query criteria matches all documents in the collection

② adding `.pretty()` expands the output

336.6. Return Only Specific Fields

We can limit the fields returned by using a "projection" expression. `1` means to include. `0` means to exclude. `_id` is automatically included and must be explicitly excluded. All other fields are automatically excluded and must be explicitly included.

Return Only Specific Fields

```
> db.books.find({}, {title:1, published:1, _id:0}) ①
{ "title" : "GWW", "published" : ISODate("1936-06-30T00:00:00Z") }
```

① find all documents and only include the `title` and `published` date

336.7. Get Document by Id

We can obtain a document by searching on any number of its fields. The following snippet locates a document by the primary key `_id` field.

Get Document By Id

```
> db.books.find({_id:ObjectId("606c82da9ef76345a2bf0b7f")})
{ "_id" : ObjectId("606c82da9ef76345a2bf0b7f"), "title" : "GWW", "author" : "MM",
"published" : ISODate("1936-06-30T00:00:00Z") }
```

336.8. Replace Document

We can replace the entire document by providing a filter and replacement document.

The snippet below filters on the `_id` field and replaces the document with a version that modifies the `title` field.

Replace Document (Found) Example

```
> db.books.replaceOne(
  { "_id" : ObjectId("606c82da9ef76345a2bf0b7f") },
  {"title" : "Gone WW", "author" : "MM", "published" : ISODate("1936-06-30T00:00:00Z") }
)

{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 } ①
```

① result document indicates a single match was found and modified

The following snippet shows a difference in the results when a match is not found for the filter.

Replacement Document (Not Found) Example

```
> db.books.replaceOne({ "_id" : "badId"}, {"title" : "Gone WW"})
{ "acknowledged" : true, "matchedCount" : 0, "modifiedCount" : 0 } ①
```

① `matchCount` and `modifiedCount` result in 0 when filter does not match anything

The following snippet shows the result of replacing the document.

Replace Document Result

```
> db.books.findOne({_id:ObjectId("606c82da9ef76345a2bf0b7f")})  
{  
  "_id" : ObjectId("606c82da9ef76345a2bf0b7f"),  
  "title" : "Gone WW",  
  "author" : "MM",  
  "published" : ISODate("1936-06-30T00:00:00Z")  
}
```

336.9. Save/Upsert a Document

We will receive an error if we issue an `insert` a second time using an `_id` that already exists.

Example Duplicate Insert Error

```
> db.books.insert({_id:ObjectId("606c82da9ef76345a2bf0b7f"), title:"Gone WW", author:  
  "MMitchell", published:ISODate("1936-06-30")})  
WriteResult({  
  "nInserted" : 0,  
  "writeError" : {  
    "code" : 11000,  
    "errmsg" : "E11000 duplicate key error collection: test.books index: _id_ dup key:  
    { _id: ObjectId('606c82da9ef76345a2bf0b7f') }",  
  }  
})
```

We will be able to insert a new document or update an existing one using the `save` command. This very useful command performs an "upsert".

Example Save/Upsert Command

```
> db.books.save({_id:ObjectId("606c82da9ef76345a2bf0b7f"), title:"Gone WW", author:  
  "MMitchell", published:ISODate("1936-06-30")}) ①  
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

① `save` command performs an upsert

336.10. Update Field

We can update specific fields in a document using one of the update commands. This is very useful when modifying large documents or when two concurrent threads are looking to increment a value in the document.

Example Update Field

```
> filter={ "_id" : ObjectId("606c82da9ef76345a2bf0b7f") } ①
> command={$set:{title : "Gone WW" } }
> db.books.updateOne( filter, command )

{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 0 }
```

① using shell to store value in variable used in command

Update Field Result

```
> db.books.findOne({_id:ObjectId("606c82da9ef76345a2bf0b7f")})
{
  "_id" : ObjectId("606c82da9ef76345a2bf0b7f"),
  "title" : "Gone WW",
  "author" : "MM",
  "published" : ISODate("1936-06-30T00:00:00Z")
}
```

336.11. Delete a Document

We can delete a document using the **delete** command and a filter.

Delete Document by Primary Key

```
> db.books.deleteOne({_id:ObjectId("606c82da9ef76345a2bf0b7f")})
{ "acknowledged" : true, "deletedCount" : 1 }
```

Chapter 337. Paging Commands

As with most `find()` implementations, we need to take care to provide a limit to the number of documents returned. The Mongo shell has a built-in default limit. We can control what the database is asked to do using a few paging commands.

337.1. Sample Documents

This example has a small collection of 10 documents.

Count Documents

```
> db.books.count({})
10
```

The following lists the primary key, title, and author. There is no sorting or limits placed on this output

Document Titles and Authors

```
> db.books.find({}, {title:1, author:1})
{ "_id" : ObjectId("607c77169fca586207a97242"), "title" : "123Pale Kings and Princes",
"author" : "Lanny Miller" }
{ "_id" : ObjectId("607c77169fca586207a97243"), "title" : "123Bury My Heart at Wounded
Knee", "author" : "Ilona Leffler" }
{ "_id" : ObjectId("607c77169fca586207a97244"), "title" : "123Carrion Comfort",
"author" : "Darci Jacobs" }
{ "_id" : ObjectId("607c77169fca586207a97245"), "title" : "123Antic Hay", "author" :
"Dorcus Harris Jr." }
{ "_id" : ObjectId("607c77169fca586207a97246"), "title" : "123Where Angels Fear to
Tread", "author" : "Latashia Gerhold" }
{ "_id" : ObjectId("607c77169fca586207a97247"), "title" : "123Tiger! Tiger!", "author"
: "Miguel Gulgowski DVM" }
{ "_id" : ObjectId("607c77169fca586207a97248"), "title" : "123Waiting for the
Barbarians", "author" : "Curtis Willms II" }
{ "_id" : ObjectId("607c77169fca586207a97249"), "title" : "123A Time of Gifts",
"author" : "Babette Grimes" }
{ "_id" : ObjectId("607c77169fca586207a9724a"), "title" : "123Blood's a Rover",
"author" : "Daryl O'Kon" }
{ "_id" : ObjectId("607c77169fca586207a9724b"), "title" : "123Precious Bane", "author"
: "Jarred Jast" }
```

337.2. `limit()`

We can limit the output provided by the database by adding the `limit()` command and supplying the maximum number of documents to return.

Example limit() Command

```
> db.books.find({}, {title:1, author:1}).limit(3) ① ② ③
{ "_id" : ObjectId("607c77169fca586207a97242"), "title" : "123Pale Kings and Princes",
"author" : "Lanny Miller" }
{ "_id" : ObjectId("607c77169fca586207a97243"), "title" : "123Bury My Heart at Wounded
Knee", "author" : "Ilona Leffler" }
{ "_id" : ObjectId("607c77169fca586207a97244"), "title" : "123Carrion Comfort",
"author" : "Darci Jacobs" }
```

- ① find all documents matching {} filter
- ② return projection of _id (default), title` , and author
- ③ limit results to first 3 documents

337.3. sort()/skip()/limit()

We can page through the data by adding the skip() command. It is common that skip() is accompanied by sort() so that the follow on commands are using the same criteria.

The following snippet shows the first few documents after sorting by author.

Paging Example, First Page

```
> db.books.find({}, {author:1}).sort({author:1}).skip(0).limit(3) ①
{ "_id" : ObjectId("607c77169fca586207a97249"), "author" : "Babette Grimes" }
{ "_id" : ObjectId("607c77169fca586207a97248"), "author" : "Curtis Willms II" }
{ "_id" : ObjectId("607c77169fca586207a97244"), "author" : "Darci Jacobs" }
```

- ① return first page of limit() size, after sorting by author

The following snippet shows the second page of documents sorted by author.

Paging Example, First Page

```
> db.books.find({}, {author:1}).sort({author:1}).skip(3).limit(3) ①
{ "_id" : ObjectId("607c77169fca586207a9724a"), "author" : "Daryl O'Kon" }
{ "_id" : ObjectId("607c77169fca586207a97245"), "author" : "Dorcas Harris Jr." }
{ "_id" : ObjectId("607c77169fca586207a97243"), "author" : "Ilona Leffler" }
```

- ① return second page of limit() size, sorted by author

The following snippet shows the last page of documents sorted by author. In this case, we have less than the limit available.

Paging Example, First Page

```
> db.books.find({}, {author:1}).sort({author:1}).skip(9).limit(3) ①
{ "_id" : ObjectId("607c77169fca586207a97247"), "author" : "Miguel Gulgowski DVM" }
```

① return last page sorted by author

Chapter 338. Aggregation Pipelines

There are times when we need to perform multiple commands and reshape documents. It may be more efficient and better encapsulated to do within the database versus issuing multiple commands to the database. MongoDB provides a feature called the [Aggregation Pipeline](#) that performs a sequence of commands called stages.

The intent of introducing the Aggregation topic is for those cases where one needs extra functionality without making multiple trips to the database and back to the client. The examples here will be very basic.

338.1. Common Commands

Some of these commands are common to `db.<collection>.find()`:

- criteria
- offset
- project
- limit
- sort

The primary difference between aggregate's use of these common commands and `find()` is that `find()` can only operate against the documents in the collection. `aggregate()` can work against the documents in the collection and any intermediate reshaping of the results along the pipeline.



Downstream Pipeline Stages do not use Collection Indexes

Only initial aggregation pipeline stage commands—operating against the database collection—can take advantage of indexes.

338.2. Unique Commands

Some commands unique to aggregation include:

- group - similar to SQL's "group by" for a JOIN, allowing us to locate distinct, common values across multiple documents and perform a group operation (like `sum`) on their remaining fields
- lookup - similar functionality to SQL's JOIN, where values in the results are used to locate additional information from other collections for the result document before returning to the client
- ...(see [Aggregate Pipeline Stages](#) documentation)

338.3. Simple Match Example

The following example implements functionality we could have implemented with `db.books.find()`. It uses 5 stages:

- `$match` - to select documents with title field containing the letter T
- `$sort` - to order documents by `author` field in descending order
- `$project` - return only the `_id` (default) and `author` fields

- **\$skip** - to skip over 0 documents
- **\$limit** - to limit output to 2 documents

Aggregate Simple Match Example

```
> db.books.aggregate([
  {$match: {title:/T/}},
  {$sort: {author:-1}},
  {$project:{author:1}},
  {$skip:0},
  {$limit:2} ])
{ "_id" : ObjectId("607c77169fca586207a97247"), "author" : "Miguel Gulgowski DVM" }
{ "_id" : ObjectId("607c77169fca586207a97246"), "author" : "Latashia Gerhold" }
```

338.4. Count Matches

This example implements a count of matching fields on the database. The functionality could have been achieved with `db.books.count()`, but it gives us a chance to show a few things that can be leveraged in more complex scenarios.

- **\$match** - to select documents with title field containing the letter T
- **\$group** - to re-organize/re-structure the documents in the pipeline to gather them under a new, primary key and to perform an aggregate function on their remaining fields. In this case we are assigning all documents the `null` primary key and incrementing a new field called `count` in the result document.

Aggregate Count Example

```
> db.books.aggregate([
  {$match:{ title:/T/}},
  {$group: {_id:null, count:{ $sum:1}}} ])
{ "_id" : null, "count" : 3 }
```

① create a new document with field `count` and increment value by 1 for each occurrence

② the resulting document is re-shaped by pipeline

The following example assigns the primary key (`_id`) field to the `author` field instead, causing each document to a distinct `author` that just happens to have only 1 instance each.

Aggregate Count Example with Unique Primary Key

```
> db.books.aggregate([
  {$match:{ title:/T/}},
  {$group: {_id:"$author", count:{ $sum:1}}} ])
{ "_id" : "Miguel Gulgowski DVM", "count" : 1 }
{ "_id" : "Latashia Gerhold", "count" : 1 }
{ "_id" : "Babette Grimes", "count" : 1 }
```

① assign primary key to `author` field

Chapter 339. Helpful Commands

This section contains a set if helpful Mongo shell commands.

339.1. Default Database

We can invoke the Mongo shell with credentials and be immediately assigned a named, default database.

- authenticating as usual
- supplying the database to execute against
- supplying the database to authenticate against (commonly `admin`)

The following snippet shows an example of authenticating as `admin` and starting with `test` as the default database for follow-on commands.

Example Set Default Database Command

```
$ docker-compose exec mongodb mongo test -u admin -p secret --authenticationDatabase
admin
...
> db.getName()
test
> show collections
books
```

339.2. Command-Line Script

We can invoke the Mongo shell with a specific command to execute by using the `--eval` command line parameter.

The following snippet shows an example of listing the contents of the `books` collection in the `test` database.

Example Script Command

```
$ docker-compose exec mongodb mongo test -u admin -p secret --authenticationDatabase
admin --eval 'db.books.find({}, {author:1})'

MongoDB shell version v4.4.0
connecting to: mongodb://127.0.0.1:27017/test?authSource=admin&compressors=disabled&g
ssapiServiceName=mongodb
Implicit session: session { "id" : UUID("47e146a5-49c0-4fe4-be67-cc8e72ea0ed9") }
MongoDB server version: 4.4.0
{ "_id" : ObjectId("607c77169fca586207a97242"), "author" : "Lanny Miller" }
{ "_id" : ObjectId("607c77169fca586207a97243"), "author" : "Ilona Leffler" }
{ "_id" : ObjectId("607c77169fca586207a97244"), "author" : "Darci Jacobs" }
{ "_id" : ObjectId("607c77169fca586207a97245"), "author" : "Dorcas Harris Jr." }
```

```
{ "_id" : ObjectId("607c77169fca586207a97246"), "author" : "Latashia Gerhold" }
{ "_id" : ObjectId("607c77169fca586207a97247"), "author" : "Miguel Gulgowski DVM" }
{ "_id" : ObjectId("607c77169fca586207a97248"), "author" : "Curtis Willms II" }
{ "_id" : ObjectId("607c77169fca586207a97249"), "author" : "Babette Grimes" }
{ "_id" : ObjectId("607c77169fca586207a9724a"), "author" : "Daryl O'Kon" }
{ "_id" : ObjectId("607c77169fca586207a9724b"), "author" : "Jarred Jast" }
```

Chapter 340. Summary

In this module, we learned:

- to identify a MongoDB collection, document, and fields
- to create a database and collection
- access a MongoDB database using the Mongo shell
- to perform basic CRUD actions on documents to manipulate a MongoDB collection
- to perform paging commands to control returned results
- to leverage the aggregation pipeline for more complex commands

MongoTemplate

copyright © 2024 jim stafford (jim.stafford@jhu.edu)

Chapter 341. Introduction

There are at least three (3) different APIs for interacting with MongoDB using Java—the last two from Spring are closely related.

MongoClient

is the core API from Mongo.

MongoOperations (interface)/MongoTemplate (implementation class)

is a command-based API around MongoClient from Spring and integrated into Spring Boot

Spring Data MongoDB Repository

is a repository-based API from Spring Data that is consistent with Spring Data JPA

This lecture covers implementing interactions with MongoDB using the MongoOperations API, implemented using MongoTemplate. Even if one intends to use the repository-based API, the MongoOperations API will still be necessary to implement various edge cases—like individual field changes versus whole document replacements.

341.1. Goals

The student will learn:

- to set up a MongoDB Maven project with references to embedded test and independent development and operational instances
- to map a POJO class to a MongoDB collection
- to implement MongoDB commands using a Spring command-level MongoOperations/MongoTemplate Java API

341.2. Objectives

At the conclusion of this lecture and related exercises, the student will be able to:

1. declare project dependencies required for using Spring's MongoOperations/MongoTemplate API
2. implement basic unit testing using a (seemingly) embedded MongoDB
3. define a connection to a MongoDB
4. switch between the embedded test MongoDB and stand-alone MongoDB for interactive development inspection
5. define a `@Document` class to map to MongoDB collection
6. inject a MongoOperations/MongoTemplate instance to perform actions on a database
7. perform whole-document CRUD operations on a `@Document` class using the Java API
8. perform surgical field operations using the Java API
9. perform queries with paging properties

10. perform Aggregation pipeline operations using the Java API

Chapter 342. MongoDB Project

Except for the possibility of indexes and defining specialized collection features — there is much less schema rigor required to bootstrap a MongoDB project or collection before using. Our primary tasks will be to

- declare a few, required dependencies
- setup project for integration testing with an embedded MongoDB instance to be able to run tests with zero administration
- conveniently switch between an embedded and stand-alone MongoDB instance to be able to inspect the database using the MongoDB shell during development

342.1. MongoDB Project Dependencies

The following snippet shows a dependency declaration for MongoDB APIs.

MongoDB Project Dependencies

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb</artifactId> ①
</dependency>
```

① brings in all dependencies required to access the database using Spring Data MongoDB

That dependency primarily brings in dependencies that are general to Spring Data and specific to MongoDB.

MongoDB Starter Dependencies

```
[INFO] +- org.springframework.boot:spring-boot-starter-data-mongodb:jar:3.3.2:compile
[INFO] |   +- org.mongodb:mongodb-driver-sync:jar:5.0.1:compile
[INFO] |   |   +- org.mongodb:bson:jar:5.0.1:compile
[INFO] |   |   \- org.mongodb:mongodb-driver-core:jar:5.0.1:compile
[INFO] |   |       \- org.mongodb:bson-record-codec:jar:5.0.1:runtime
[INFO] |   \- org.springframework.data:spring-data-mongodb:jar:4.3.2:compile
[INFO] |       +- org.springframework:spring-tx:jar:6.1.11:compile
[INFO] |       \- org.springframework.data:spring-data-commons:jar:3.3.2:compile
```

That is enough to cover integration with an external MongoDB during operational end-to-end scenarios. Next, we need to address the integration test environment.

342.2. MongoDB Project Integration Testing Options

MongoDB, like Postgres, is not written in Java (MongoDB is [written in C++](#)). That means that we cannot simply instantiate MongoDB within our integration test JVM. We have at least four options:

- Mocks
- [Fongo](#) in-memory MongoDB implementation (dormant)
- [Flapdoodle](#) embedded MongoDB (alive) and Auto Configuration or referenced Maven plugins (dormant):
 - [maven-mongodb-plugin](#)
 - [embedmongo-maven-plugin](#)
- [Testcontainers](#) Docker wrapper ([active](#))

Each should be able to do the job for what we want to do here. However,

- With nothing else in place, Mocks would be an option for trivial cases. However, not an option to test out database client code.
- Although Fongo is an in-memory solution, it is not MongoDB and edge cases may not work the same as a real MongoDB instance. Most of the work on this option was developed in 2014, slowed by 2018, and the last commit to master was ~2020.
- Flapdoodle calls itself "embedded". However, the term embedded is meant to mean "within the scope of the test" and not "within the process itself". The download and management of the server is what is embedded. The [Spring Boot 2.7 Documentation](#) discusses Flapdoodle, and the [Spring Boot 2.7 Embedded Mongo AutoConfiguration](#) seamlessly integrates Flapdoodle with a few options. However, Spring Boot dropped direct support for Flapdoodle in 3.0. Flapdoodle is still actively maintained and can still be used with Spring Boot 3 (as I will show in the examples). Anything Spring Boot no longer supplies for the integration (e.g., AutoConfiguration, ConfigurationProperties) has now moved over to Flapdoodle.

Full control of the configuration can be performed using the referenced Maven plugins or writing your own [@Configuration](#) beans that invoke the Flapdoodle API directly.

- Testcontainers provides full control over the versions and configuration of MongoDB instances using Docker. The following [article](#) points out some drawback to using Flapdoodle and how leveraging Testcontainers solved their issues.^[1] Spring Boot dropped their direct support for Flapdoodle in version 3 in favor of using Testcontainers. I showed how to easily integrate Testcontainers into an integration test earlier in the lectures. This legacy example will continue to use Flapdoodle to show the still viable legacy option.

342.3. Flapdoodle Test Dependencies

This lecture will use the Flapdoodle Embedded Mongo Spring 3.x setup. The following [Maven dependency](#) will bring in Flapdoodle libraries and trigger the Spring Boot Embedded MongoDB Auto Configuration

Flapdoodle Test Dependencies

```
<dependency>
  <groupId>de.flapdoodle.embed</groupId>
  <artifactId>de.flapdoodle.embed.mongo.spring30x</artifactId>
  <scope>test</scope>
```

```
</dependency>
```

```
[INFO] +- de.flapdoodle.embed:de.flapdoodle.embed.mongo.spring30x:jar:4.11.0:test
[INFO] | \- de.flapdoodle.embed:de.flapdoodle.embed.mongo:jar:4.11.1:test
...
...
```

A test instance of MongoDB is downloaded and managed through a test library called [Flapdoodle Embedded Mongo](#). It is called "embedded", but unlike H2 and other embedded Java RDBMS implementations—the only thing embedded about this capability is the logical management feel. Technically, the library downloads a MongoDB instance (cached), starts, and stops the instance as part of running the test. A Flapdoodle AutoConfigure will activate Flapdoodle when running a unit integration test, and it detects the library on the classpath. We can bypass the use of Flapdoodle and use an externally managed MongoDB instance by turning off the Flapdoodle starter.

Same AutoConfigure, Different Artifact/Source



The Flapdoodle AutoConfiguration used to be [included within Spring Boot \(≤ 2.7\)](#) and has now moved to [Flapdoodle](#). It is basically the same class/purpose. It just now requires a direct dependency on a Flapdoodle artifact versus getting it through [spring-boot-starter-mongodb](#).

342.4. Flapdoodle Properties

You must set the `de.flapdoodle.mongodb.embedded.version` property to a supported version in your test properties.

Required Version Property

```
de.flapdoodle.mongodb.embedded.version=4.4.0
```

Otherwise, you will likely get the following error when trying to start tests.

Missing Required Property Error

```
IllegalStateException: Set the de.flapdoodle.mongodb.embedded.version property or
define your own IFeatureAwareVersion bean to use embedded MongoDB
```

342.5. MongoDB Access Objects

There are two primary beans of interest when we connect and interact with MongoDB: MongoClient and MongoOperations/MongoTemplate.

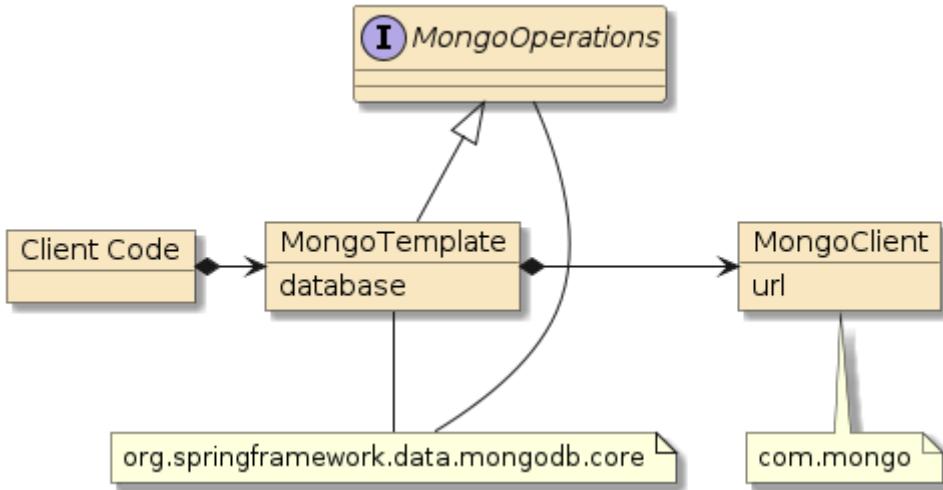


Figure 152. Injectable MongoDB Access Objects

- [MongoClient](#) is a client provided by Mongo that provides the direct connection to the database and mimics the behavior of the Mongo Shell using Java. AutoConfiguration will automatically instantiate this, but can be customized using [MongoClients](#) factory class.
- [MongoOperations](#) is an interface provided by Spring Data that defines a type-mapped way to use the client
- [MongoTemplate](#) is the implementation class for MongoOperations—also provided by Spring Data. AutoConfiguration will automatically instantiate this using the [MongoClient](#) and a specific database name.

Embedded MongoDB Auto Configuration Instantiates MongoClient to Reference Flapdoodle Instance



By default, the Embedded MongoDB Auto Configuration class will instantiate a MongoDB instance using Flapdoodle and instantiate a MongoClient that references that instance. The Embedded MongoDB Auto Configuration class must be explicitly disabled if you wish to use a different MongoDB target. This will be demonstrated in this lecture.

342.6. MongoDB Connection Properties

To communicate with an explicit MongoDB server, we need to supply various properties or combine them into a single [spring.data.mongodb.uri](#)

The following example property file lists the individual properties commented out and the combined properties expressed as a URL. These will be used to automatically instantiate an injectable [MongoClient](#) and [MongoTemplate](#) instance to a remote (non-Flapdoodle) instance.

application-mongodb.properties

```
#spring.data.mongodb.host=localhost
#spring.data.mongodb.port=27017
#spring.data.mongodb.database=test
#spring.data.mongodb.authentication-database=admin
#spring.data.mongodb.username=admin
```

```
#spring.data.mongodb.password=secret  
spring.data.mongodb.uri=mongodb://admin:secret@localhost:27017/test?authSource=admin
```

342.7. Injecting MongoTemplate

The MongoDB starter takes care of declaring key `MongoClient` and `MongoTemplate @Bean` instances that can be injected into components. Generally, injection of the `MongoClient` will not be necessary.

MongoTemplate Class Injection

```
@Autowired  
private MongoTemplate mongoTemplate; ①
```

① `MongoTemplate` defines a starting point to interface to MongoDB in a Spring application

Alternatively, we can inject using the interface of `MongoTemplate`.

Alternate Interface Injection

```
@Autowired  
private MongoOperations mongoOps; ①
```

① `MongoOperations` is the interface for `MongoTemplate`

342.8. Disabling Embedded MongoDB

By default, our tests get configured to use the Embedded MongoDB, Flapdoodle test instance. For development, we may want to work against a live MongoDB instance so we can interactively inspect the database using the Mongo shell. The only way to prevent using Embedded MongoDB during testing—is to disable the starter.

The following snippet shows the command-line system property that will disable `EmbeddedMongoAutoConfiguration` from activating. That will leave only the standard `MongoAutoConfiguration` to execute and setup `MongoClient` using `spring.data.mongodb` properties.

Disable Embedded Mongo using Command Line System Property

```
-Dspring.autoconfigure.exclude=\  
de.flapdoodle.embed.mongo.spring.autoconfigure.EmbeddedMongoAutoConfiguration
```

To make things simpler, I added a conditional `@Configuration` class that would automatically trigger the exclusion of the `EmbeddedMongoAutoConfiguration` when the `spring.data.mongodb.uri` was present.

Disable Embedded Mongo using Conditional @Configuration Class

```
import de.flapdoodle.embed.mongo.spring.autoconfigure.EmbeddedMongoAutoConfiguration;  
...  
@Configuration(proxyBeanMethods = false)
```

```

@ConditionalOnProperty(prefix="spring.data.mongodb", name="uri", matchIfMissing=false)①
@EnableAutoConfiguration(exclude = EmbeddedMongoAutoConfiguration.class) ②
public class DisableEmbeddedMongoConfiguration { }

```

① class is activated on the condition that property `spring.data.mongodb.uri` be present

② when activated, class definition will disable `EmbeddedMongoAutoConfiguration`

342.9. @ActiveProfiles

With the ability to turn on/off the `EmbeddedMongo` and `MongoDB` configurations, it would be nice to make this work seamlessly with profiles. We know that we can define an `@ActiveProfiles` for integration tests to use, but this is very static. It cannot be changed during normal build time using a command-line option.

Static Active Profiles Declaration

```

@SpringBootTest(classes= NTestConfiguration.class) ①
@ActiveProfiles(profiles="mongodb") ②
public class MongoOpsBooksNTest {

```

① defines various injectable instances for testing

② statically defines which profile will be currently active

To make this more flexible at runtime (from the Maven/Surefire command line), we can take advantage of the `resolver` option of `@ActiveProfiles`. Anything we list in `profiles` is the default. Anything that is returned from the `resolver` is what is used instead. The `resolver` is an instance of `ActiveProfilesResolver`.

Dynamic Active Profile Determination

```

@SpringBootTest(classes= NTestConfiguration.class)
@ActiveProfiles(profiles={ ... }, resolver = ...class) ① ②
public class MongoOpsBooksNTest {

```

① `profiles` list the default profile(s) to use

② `resolver` implements `ActiveProfilesResolver` and determines what profiles to use at runtime

342.10. TestProfileResolver

I implemented a simple class based on an example from the internet from Amit Kumar.^[2] The class will inspect the `spring.profiles.active` if present and return an array of strings containing those profiles. If the property does not exist, then the default options of the test class are used.

Manual Specification of Active Profiles

```
-Dspring.profiles.active=mongodb,foo,bar
```

The following snippet shows how that is performed.

@ActiveProfile resolver class

```
import org.springframework.test.context.ActiveProfilesResolver;
import org.springframework.test.context.support.DefaultActiveProfilesResolver;
...
//Ref: https://www.allprogrammingtutorials.com/tutorials/overriding-active-profile-
boot-integration-tests.php
public class TestProfileResolver implements ActiveProfilesResolver {
    private final String PROFILE_KEY = "spring.profiles.active";
    private final ActiveProfilesResolver defaultResolver = new
DefaultActiveProfilesResolver();

    @Override
    public String[] resolve(Class<?> testClass) {
        return System.getProperties().containsKey(PROPERTY_KEY) ?
            //return profiles expressed in property as array of strings
            System.getProperty(PROPERTY_KEY).split("\\s*,\\s*") : ①
            //return profile(s) expressed in the class' annotation
            defaultResolver.resolve(testClass);
    }
}
```

① regexp splits string at the comma (',') character and an unlimited number of contiguous whitespace characters on either side

The `resolver` can get the expressed defaults from an instance of the `DefaultActiveProfilesResolver` class, which uses reflection to search for the `@ActiveProfile` annotation on the provided `testClass`.

342.11. Using TestProfileResolver

The following snippet shows how `TestProfileResolver` can be used by an integration test.

- The test uses no profile by default — activating Embedded MongoDB.
- If the `mongodb` profile is specified using a system property or temporarily inserted into the source — then that profile will be used.
- Since my `mongodb` profile declares `spring.data.mongodb.uri`, Embedded MongoDB is deactivated.

Example Use of TestProfileResolver

```
@SpringBootTest(classes= NTestConfiguration.class) ①
@ActiveProfiles(resolver = TestProfileResolver.class) ②
//@ActiveProfiles(profiles="mongodb", resolver = TestProfileResolver.class) ③
public class MongoOpsBooksNTest {
```

① defines various injectable instances for testing

- ② defines which profile will be currently active
- ③ defines which profile will be currently active, with `mongodb` being the default profile

342.12. Inject MongoTemplate

In case you got a bit lost in that testing detour, we are now at a point where we can begin interacting with our chosen MongoDB instance using an injected `MongoOperations` (the interface) or `MongoTemplate` (the implementation class).

Inject MongoTemplate

```
@AutoConfigure  
private MongoTemplate mongoTemplate;
```

I wanted to show you how to use the running MongoDB when we write the integration tests using `MongoTemplate` so that you can inspect the live DB instance with the Mongo shell while the database is undergoing changes. Refer to the previous MongoDB lecture for information on how to connect the DB with the Mongo shell.

342.13. Testcontainers

I did not include coverage of Testcontainers/Mongo in this lecture. This is because I felt the setup and connection topic was covered good enough during the [Testcontainers lecture](#), wanted to keep a demonstration of Flapdoodle, and to prevent topic/dependency overload. Testcontainers specifically replaces the Embedded MongoDB, Flapdoodle support and is just another way to launch the external MongoDB Docker container used within this lecture example.

[1] "Fast and stable MongoDB-based tests in Spring", Piotr Kubowicz, Dec 2020

[2] "Overriding Active Profiles in Spring Boot Integration Tests", Amit Kumar, 2018

Chapter 343. Example POJO

We will be using an example `Book` class to demonstrate some database mapping and interaction concepts. The class properties happen to be mutable, and the class provides an all-arg constructor to support a builder and adds `with()` modifiers to be able to chain modifications (using new instances). These are not specific requirements of Spring Data Mongo. Spring Data Mongo is designed to work with many different POJO designs.

MongoTemplate Works with Classes lacking No-Arg Constructor



This example happens to have no no-arg constructor. The default no-arg constructor is disabled by the all-arg constructor declared by both `@Builder` (package-friendly access) and `@AllArgsConstructor` (public access).

Book POJO being mapped to database

```
package info.ejava.examples.db.mongo.books.bo;  
...  
import org.springframework.data.annotation.Id;  
import org.springframework.data.mongodb.core.mapping.Document;  
import org.springframework.data.mongodb.core.mapping.Field;  
  
@Document(collection = "books")  
@Getter  
@Setter  
@Builder  
@With  
@AllArgsConstructor  
public class Book {  
    @Id @Setter(AccessLevel.NONE)  
    private String id;  
    @Field(name="title")  
    private String title;  
    private String author;  
    private LocalDate published;  
}
```

343.1. Property Mapping

343.1.1. Collection Mapping

Spring Data Mongo will map instances of the class to a collection

- by the same name as the class (e.g., `book`, by default)
- by the `collection` name supplied in the `@Document` annotation

Collection Mapping

```
@Document(collection = "books") ②
public class Book { ①}
```

① instances are, by default, mapped to the "book" collection

② `@Documentation.collection` annotation property overrides default collection name

MongoTemplate also provides the ability to independently provide the collection name during the command—which makes the class mapping even less important.

343.1.2. Primary Key Mapping

The MongoDB `_id` field will be mapped to a field that either

- is called `id`
- is annotated with `@Id` (`@org.springframework.data.annotation.Id`; this is not the same as `@jakarta.persistence.Id` used with JPA)
- is mapped to field `_id` using `@Field` annotation

Primary Key Mapping

```
import org.springframework.data.annotation.Id;

@Id ①
private String id; ① ②
```

① property is both named `id` and annotated with `@Id` to map to `_id` field

② `String` `id` type can be mapped to auto-generated MongoDB `_id` field

Only `_id` fields mapped to `String`, `BigInteger`, or `ObjectId` can have auto-generated `_id` fields mapped to them.

343.2. Field Mapping

Class properties will be mapped by default to a field of the same name. The `@Field` annotation can be used to customize that behavior.

Field Mapping

```
import org.springframework.data.mongodb.core.mapping.Field;

@Field(name="title") ①
private String titleXYZ;
```

① maps Java property `titleXYZ` to MongoDB document field `title`

We can annotate a property with `@Transient` to prevent a property from being stored in the

database.

Transient Mapping

```
import org.springframework.data.annotation.Transient;  
  
① @Transient ①  
private String dontStoreMe;
```

① `@Transient` excludes the Java property from being mapped to the database

343.3. Instantiation

Spring Data Mongo leverages constructors in the [following order](#)

1. No argument constructor
2. Multiple argument constructor annotated with `@PersistenceConstructor`
3. Solo, multiple argument constructor (preferably an all-args constructor)

Given our example, the all-args constructor will be used.

343.4. Property Population

For properties not yet set by the constructor, Spring Data Mongo will set fields using the [following order](#)

1. use `setter()` if supplied
2. use `with()` if supplied, to construct a copy with the new value
3. directly modify the field using reflection

Chapter 344. Command Types

MongoTemplate offers different types of command interactions

Whole Document

complete document passed in as argument and/or returned as a result

By Id

command performed on document matching provided ID

Filter

command performed on documents matching filter

Field Modifications

command makes field level changes to database documents

Paging

options to finder commands to limit results returned

Aggregation Pipeline

sequential array of commands to be performed on the database

These are not the only categories of commands you could come up with describing the massive set, but it will be enough to work with for a while. Inspect the [MongoTemplate Javadoc](#) for more options and detail.

Chapter 345. Whole Document Operations

The `MongoTemplate` instance already contains a reference to a specific database and the `@Document` annotation of the POJO has the collection name — so the commands know exactly which collection to work with. Commands also offer options to express the collection as a string at command-time to add flexibility to mapping approaches.

345.1. insert()

`MongoTemplate` offers an explicit `insert()` that will always attempt to insert a new document without checking if the ID already exists. If the created document has a generated ID not yet assigned — then this should always successfully add a new document.

One thing to note about class mapping is that `MongoTemplate` adds an additional field to the document during insert. This field is added to support polymorphic instantiation of result classes.

MongoTemplate _class Field

```
{ "_id" : ObjectId("608b3021bd49095dd4994c9d"),
  "title" : "Vile Bodies",
  "author" : "Ernesto Rodriguez",
  "published" : ISODate("2015-03-10T04:00:00Z"),
  "_class" : "info.ejava.examples.db.mongo.books.bo.Book" } ①
```

① `MongoTemplate` adds extra `_class` field to help dynamically instantiate query results

This behavior can be turned off by configuring your own instance of `MongoTemplate` and following the example from Mkyong.com in this [link](#).

345.1.1. insert() Successful

The following snippet shows an example of a transient book instance being successfully inserted into the database collection using the `insert` command.

MongoTemplate insert() Successful

```
//given an entity instance
Book book = ...
//when persisting
mongoTemplate.insert(book); ① ②
//then documented is persisted
then(book.getId()).isNotNull();
then(mongoTemplate.findById(book.getId(), Book.class)).isNotNull();
```

① transient document assigned an ID and inserted into database collection

② database referenced by `MongoTemplate` and collection identified in `Book @Document.collection` annotation

345.1.2. insert() Duplicate Fail

If the created document is given an assigned ID value, then the call will fail with a `DuplicateKeyException` exception if the ID already exists.

MongoTemplate create() with Duplicate Key Throws Exception

```
import org.springframework.dao.DuplicateKeyException;  
...  
//given a persisted instance  
Book book = ...  
mongoTemplate.insert(book);  
//when persisting an instance by the same ID  
Assertions.assertThrows(DuplicateKeyException.class,  
    ()->mongoTemplate.insert(book)); ①
```

① document with ID matching database ID cannot be inserted

345.2. save()/Upsert

The `save()` command is an "upsert" (Update or Insert) command and likely the simplest form of "upsert" provided by `MongoTemplate` (there are more). It can be used to insert a document if new or replace if already exists - based only on the evaluation of the ID.

345.2.1. Save New

The following snippet shows a new transient document being saved to the database collection. We know that it is new because the ID is unassigned and generated at `save()` time.

Upsert Example - Save New

```
//given a document not yet saved to DB  
Book transientBook = ...  
assertThat(transientBook.getId()).isNull();  
//when - updating  
mongoTemplate.save(transientBook);  
//then - db has new state  
then(transientBook.getId()).isNotNull();  
Book dbBook = mongoTemplate.findById(transientBook.getId());  
then(dbBook.getTitle()).isEqualTo(transientBook.getTitle());  
then(dbBook.getAuthor()).isEqualTo(transientBook.getAuthor());  
then(dbBook.getPublished()).isEqualTo(transientBook.getPublished());
```

345.2.2. Replace Existing

The following snippet shows a new document instance with the same ID as a document in the database, but with different values. In this case, `save()` performs an update/(whole document replacement).

UpsertExample - Replace Existing

```
//given a persisted instance
Book originalBook = ...
mongoTemplate.insert(originalBook);
Book updatedBook = mapper.map(dtoFactory.make()).withId(originalBook.getId());
assertThat(updatedBook.getTitle()).isNotEqualTo(originalBook.getTitle());
//when - updating
mongoTemplate.save(updatedBook);
//then - db has new state
Book dbBook = mongoTemplate.findById(book.getId(), Book.class);
then(dbBook.getTitle()).isEqualTo(updatedBook.getTitle());
then(dbBook.getAuthor()).isEqualTo(updatedBook.getAuthor());
then(dbBook.getPublished()).isEqualTo(updatedBook.getPublished());
```

345.3. remove()

`remove()` is another command that accepts a document as its primary input. It returns some metrics about what was found and removed.

The snippet below shows the successful removal of an existing document. The `DeleteResult` response document provides feedback of what occurred.

Successful Remove Example

```
//given a persisted instance
Book book = ...
mongoTemplate.save(book);
//when - deleting
DeleteResult result = mongoTemplate.remove(book);
long count = result.getDeletedCount();
//then - no longer in DB
then(count).isEqualTo(1);
then(mongoTemplate.findById(book.getId(), Book.class)).isNotNull();
```

Chapter 346. Operations By ID

There are very few commands that operate on an explicit ID. `findById` is the only example. I wanted to highlight the fact that most commands use a flexible query filter, and we will show examples of that in after introducing `ById`.

346.1. `findById()`

`findById()` will return the complete document associated with the supplied ID.

The following snippet shows an example of the document being found.

findById() Found Example

```
//given a persisted instance
Book book = ...
//when finding
Book dbBook = mongoTemplate.findById(book.getId(), Book.class); ①
//then document is found
then(dbBook.getId()).isEqualTo(book.getId());
then(dbBook.getTitle()).isEqualTo(book.getTitle());
then(dbBook.getAuthor()).isEqualTo(book.getAuthor());
then(dbBook.getPublished()).isEqualTo(book.getPublished());
```

① `Book` class is supplied to identify the collection and the type of response object to populate

A missing document does not throw an exception — just returns a null object.

findById() Not Found Example

```
//given a persisted instance
String missingId = "12345";
//when finding
Book dbBook = mongoTemplate.findById(missingId, Book.class);
//then
then(dbBook).isNull();
```

Chapter 347. Operations By Query Filter

Many commands accept a `Query` object used to filter which documents in the collection the command applies to. The `Query` can express:

- criteria
- targeted types
- paging

We will stick to just simple the criteria here.

Example Criteria syntax

```
Criteria filter = Criteria.where("field1").is("value1")
    .and("field2").not().is("value2");
```

If we specify the collection name (e.g., "books") in the command versus the type (e.g., `Book` class), we lack the field/type mapping information. That means we must explicitly name the field and use the type known by the MongoDB collection.

Collection Name versus Mapped Type ID Expressions

```
Query.query(Criteria.where("id").is(id));           //Book.class ①
Query.query(Criteria.where("_id").is(new ObjectId(id))); // "books" ②
```

① can use property values when supplying mapped class in full command

② must supply a field and explicit mapping type when supplying collection name in full command

347.1. exists() By Criteria

`exists()` accepts a `Query` and returns a simple true or false. The query can be as simple or complex as necessary.

The following snippet looks for documents with a matching ID.

exists() By Criteria

```
//given a persisted instance
Book book = ...
mongoTemplate.save(book);
//when testing exists
Query filter = Query.query(Criteria.where("id").is(id));
boolean exists = mongoTemplate.exists(filter, Book.class);
//then document exists
then(exists).isTrue();
```

MongoTemplate was smart enough to translate the "id" property to the `_id` field and the String

value to an `ObjectId` when building the criteria with a mapped class.

MongoTemplate Generated Criteria Document

```
{ "_id" : { "$oid" : "608ae2939f024c640c3b1d4b"}}
```

347.2. delete()

`delete()` is another command that can operate on a criteria filter.

```
//given a persisted instance
Book book = ...
mongoTemplate.save(book);
//when - deleting
Query filter = Query.query(Criteria.where("id").is(id));
DeleteResult result = mongoTemplate.remove(filter, Book.class);
//then - no long in DB
then(count).isEqualTo(1);
then(mongoTemplate.existsById(book.getId())).isFalse();
```

Chapter 348. Field Modification Operations

For cases with large documents—where it would be an unnecessary expense to retrieve the entire document and then to write it back with changes—MongoTemplate can issue individual field commands. This is also useful in concurrent modifications where one wants to upsert a document (and have only a single instance) but also update an existing document with fresh information (e.g., increment a counter, set a processing timestamp)

348.1. update() Field(s)

The `update()` command can be used to perform actions on individual fields. The following example changes the title of the first document that matches the provided criteria. [Update commands](#) can have a minor complexity to include incrementing, renaming, and moving fields—as well as manipulating arrays.

update() Fields Example

```
//given a persisted instance
Book originalBook = ...
mongoTemplate.save(originalBook);
String newTitle = "X" + originalBook.getTitle();
//when - updating
Query filter = Query.query(Criteria.where("_id").is(new ObjectId(id)));①
Update update = new Update(); ②
update.set("title", newTitle); ③
UpdateResult result = mongoTemplate.updateFirst(filter, update, "books"); ④
//{ "_id" : { "$oid" : "60858ca8a3b90c12d3bb15b2"}},
//{ "$set" : { "title" : "XTo Sail Beyond the Sunset"}}
long found = result.getMatchedCount();
//then - db has new state
then(found).isEqualTo(1);
Book dbBook = mongoTemplate.findById(originalBook.getId());
then(dbBook.getTitle()).isEqualTo(newTitle);
then(dbBook.getAuthor()).isEqualTo(originalBook.getAuthor());
then(dbBook.getPublished()).isEqualTo(originalBook.getPublished());
```

① identifies a criteria for update

② individual commands to apply to the database document

③ document found will have its `title` changed

④ must use explicit `_id` field and `ObjectId` value when using ("books") collection name versus `Book` class

348.2. upsert() Fields

If the document was not found and we want to be in a state where one will exist with the desired title, we could use an `upsert()` instead of an `update()`.

upsertFields() Example

```
UpdateResult result = mongoTemplate.upsert(filter, update, "books"); ①
```

① **upsert** guarantees us that we will have a document in the books collection with the intended modifications

Chapter 349. Paging

In conjunction with `find` commands, we need to soon look to add paging instructions to sort and slice up the results into page-sized bites. `RestTemplate` offers two primary ways to express paging

- Query configuration
- Pagable command parameter

349.1. skip() / limit()

We can express offset and limit on the `Query` object using `skip()` and `limit()` builder methods.

skip() and limit()

```
Query query = new Query().skip(offset).limit(limit);
```

In the example below, a `findOne()` with `skip()` is performed to locate a single, random document.

Find Random Document

```
private final SecureRandom random = new SecureRandom();
public Optional<Book> random() {
    Optional randomSong = Optional.empty();
    long count = mongoTemplate.count(new Query(), "books");

    if (count!=0) {
        int offset = random.nextInt((int)count);
        Book song = mongoTemplate.findOne(new Query().skip(offset), Book.class); ① ②
        randomSong = song==null ? Optional.empty() : Optional.of(song);
    }
    return randomSong;
}
```

① `skip()` is eliminating offset documents from the results

② `findOne()` is reducing the results to a single (first) document

We could have also expressed the command with `find()` and `limit(1)`.

find() with limit()

```
mongoTemplate.find(new Query().skip(offset).limit(1), Book.class);
```

349.2. Sort

With offset and limit, we often need to express sort — which can get complex. Spring Data defines a `Sort` class that can express a sequence of properties to sort in ascending and/or descending order. That too can be assigned to the `Query` instance.

Sort Example

```
public List<Book> find(List<String> order, int offset, int limit) {  
    Query query = new Query();  
    query.with( Sort.by(order.toArray(new String[0]))) ; ①  
    query.skip(offset); ②  
    query.limit(limit); ③  
    return mongoTemplate.find(query, Book.class);  
}
```

- ① Query accepts a standard `Sort` type to implement ordering
- ② Query accepts a `skip` to perform an offset into the results
- ③ Query accepts a `limit` to restrict the number of returned results.

349.3. Pageable

Spring Data provides a `Pageable` type that can express sort, offset, and limit — using `Sort`, `pageSize`, and `pageNumber`. That too can be assigned to the `Query` instance.

```
int pageNo=1;  
int pageSize=3;  
Pageable pageable = PageRequest.of(pageNo, pageSize,  
                                   Sort.by(Sort.Direction.DESC, "published"));  
  
public List<Book> find(Pageable pageable) {  
    return mongoTemplate.find(new Query().with(pageable), Book.class); ①  
}
```

- ① Query accepts a `Pageable` to permit flexible ordering, offset, and limit

Chapter 350. Aggregation

Most queries can be performed using the database `find()` commands. However, as we have seen in the MongoDB lecture—some complex queries require different stages and command types to handle selections, projections, grouping, etc. For those cases, Mongo provides the Aggregation Pipeline—which can be accessed through the `MongoTemplate`.

The following snippet shows a query that locates all documents that contain an `author` field and match a regular expression.

Example Aggregation Pipeline Call

```
//given
int minLength = ...
Set<String> ids = savedBooks.stream() ... //return IDs of docs matching criteria
String expression = String.format("^.{%d,}$", minLength);
//when pipeline executed
Aggregation pipeline = Aggregation.newAggregation(
    Aggregation.match(Criteria.where("author").regex(expression)),
    Aggregation.match(Criteria.where("author").exists(true))
);
AggregationResults<Book> result = mongoTemplate.aggregate(pipeline, "books", Book.class
);
List<Book> foundSongs = result.getMappedResults();
//then expected IDs found
Set<String> foundIds = foundSongs.stream()
    .map(s->s.getId()).collect(Collectors.toSet());
then(foundIds).isEqualTo(ids);
```

Mongo BasicDocument Issue with \$exists Command

Aggregation Pipeline was forced to be used in this case, because a normal collection `find()` command was not able to accept an `exists` command with another command for that same field.

`Criteria.where("author").regex(expression).and("author").exists(true))`



`org.springframework.data.mongodb.InvalidMongoDbApiUsageException: Due to limitations of the com.mongodb.BasicDocument, you can't add a second 'author' expression specified as 'author : Document{{$exists=true}}'. Criteria already contains 'author : ^.{22,}$'.`

This provides a good example of how to divide up the commands into independent queries using Aggregation Pipeline.

Chapter 351. ACID Transactions

Before we leave the accessing MongoDB through the [MongoTemplate](#) Java API topic, I wanted to lightly cover ACID transactions.

- Atomicity
- Consistency
- Isolation
- Durability

351.1. Atomicity

MongoDB has made a lot of great strides in scale and performance by providing flexible document structures. Individual caller commands to change a document represent separate, atomic transactions. Documents can be as large or small as one desires and should take document atomicity into account when forming document schema.

However, as of MongoDB 4.0, MongoDB supports multi-document atomic transactions if absolutely necessary. The following [online resource](#) provides some background on how to achieve this. ^[1]



MongoDB Multi-Document Transactions and not the Normal Path

Just because you can implement multi-document atomic transactions and references between documents, don't default to a RDBMS mindset when designing document schema. Try to make a single document represent state that is essential to be in a consistent state.

MongoDB [documentation](#) does warn against its use. So multi-document acid transactions should not be a first choice.

351.2. Consistency

Since MongoDB does not support a fixed schema or enforcement of foreign references between documents, there is very little for the database to keep consistent. The primary consistency rules the database must enforce are any unique indexes — requiring that specific fields be unique within the collection.

351.3. Isolation

Within the context of a single document change — MongoDB ^[2]

- will always prevent a reader from seeing partial changes to a document.
- will provide a reader a complete version of a document that may have been inserted/updated after a [find\(\)](#) was initiated but before it was returned to the caller (i.e., can receive a document that no longer matches the original query)
- may miss including documents that satisfy a query after the query was initiated but before the

results are returned to the caller

351.4. Durability

The durability of a MongoDB transaction is a function of the number of nodes within a cluster that acknowledge a change before returning the call to the client. **UNACKNOWLEDGED** is fast but extremely unreliable. Other ranges, including **MAJORITY** at least guarantee that one or more nodes in the cluster have written the change. These are expressed using the MongoDB [WriteConcern](#) class.

MongoTemplate [allows us](#) to set the WriteConcern for follow-on [MongoTemplate](#) commands.

Durability is a more advanced topic and requires coverage of system administration and cluster setup—which is well beyond the scope of this lecture. My point of bringing this and other ACID topics up here is to only point out that the [MongoTemplate](#) offers access to these additional features.

[1] "Spring Data MongoDB Transactions", baeldung, 2020

[2] "Read Isolation, Consistency, and Recency", MongoDB Manual, Version 4.4

Chapter 352. Summary

In this module, we learned to:

- setup a MongoDB Maven project
- inject a MongoOperations/MongoTemplate instance to perform actions on a database
- instantiate a (seemingly) embedded MongoDB connection for integration tests
- instantiate a stand-alone MongoDB connection for interactive development and production deployment
- switch between the embedded test MongoDB and stand-alone MongoDB for interactive development inspection
- map a `@Document` class to a MongoDB collection
- implement MongoDB commands using a Spring command-level MongoOperations/MongoTemplate Java API
- perform whole-document CRUD operations on a `@Document` class using the Java API
- perform surgical field operations using the Java API
- perform queries with paging properties
- perform Aggregation pipeline operations using the Java API

Spring Data MongoDB Repository

copyright © 2024 jim stafford (jim.stafford@jhu.edu)

Chapter 353. Introduction

MongoTemplate provided a lot of capability to interface with the database, but with a significant amount of code required. Spring Data MongoDB Repository eliminates much of the boilerplate code for the most common operations and allows us access to MongoTemplate for the harder edge-cases.



Due to the common Spring Data framework between the two libraries and the resulting similarity between Spring Data JPA and Spring Data MongoDB repositories, this lecture is about 95% the same as the Spring Data JPA lecture. Although it is presumed that the Spring Data JPA lecture precedes this lecture—it was written so that was not a requirement. If you have already mastered Spring Data JPA Repositories, you should be able to quickly breeze through this material because of the significant similarities in concepts and APIs.

353.1. Goals

The student will learn:

- to manage objects in the database using the Spring Data MongoDB Repository
- to leverage different types of built-in repository features
- to extend the repository with custom features when necessary

353.2. Objectives

At the conclusion of this lecture and related exercises, the student will be able to:

1. declare a [MongoRepository](#) for an existing [@Document](#)
2. perform simple CRUD methods using provided repository methods
3. add paging and sorting to query methods
4. implement queries based on POJO examples and configured matchers
5. implement queries based on predicates derived from repository interface methods
6. implement a custom extension of the repository for complex or compound database access

Chapter 354. Spring Data MongoDB Repository

Spring Data MongoDB provides repository support for `@Document`-based mappings.^[1] We start off by writing no mapping code—just interfaces associated with our `@Document` and primary key type—and have Spring Data MongoDB implement the desired code. The Spring Data MongoDB interfaces are layered—offering useful tools for interacting with the database. Our primary `@Document` types will have a repository interface declared that inherit from `MongoRepository` and any custom interfaces we optionally define.



Figure 153. Spring Data MongoDB Repository Interfaces

The extends path was modified some with the latest version of Spring Data Commons, but the `MongoRepository` ends up being mostly the same by the time the interfaces get merged at the bottom of the inheritance tree.

[1] "Spring Data MongoDB - Reference Documentation"

Chapter 355. Spring Data MongoDB Repository Interfaces

As we go through these interfaces and methods, please remember that all of the method implementations of these interfaces (except for custom) will be provided for us.

Repository<T, ID>

marker interface capturing the `@Document` class and primary key type. Everything extends from this type.

CrudRepository<T, ID>

depicts many of the CRUD capabilities we demonstrated with the MongoOps DAO in previous MongoTemplate lecture

PagingAndSortingRepository<T, ID>

Spring Data MongoDB provides some nice end-to-end support for sorting and paging. This interface adds some sorting and paging to the `findAll()` query method provided in `CrudRepository`.

ListPagingAndSortingRepository<T, ID>

overrides the PagingAndSorting-based `Iterable<T>` return type to be a `List<T>`

ListCrudRepository

overrides all CRUD-based `Iterable<T>` return types with `List<T>`

QueryByExampleExecutor<T>

provides query-by-example methods that use prototype `@Document` instances and configured matchers to locate matching results

MongoRepository<T, ID>

brings together the `CrudRepository`, `PagingAndSortingRepository`, and `QueryByExampleExecutor` interfaces and adds several methods of its own. The methods declared are mostly generic to working with repositories—only the `insert()` methods have any specific meaning to MongoDB.

BooksRepositoryCustom/ BooksRepositoryCustomImpl

we can write our own extensions for complex or compound calls—while taking advantage of an `MongoTemplate` and existing repository methods. This allows us to encapsulate details of `update()` methods and Aggregation Pipeline as well as other MongoTemplate interfaces like GridFS and Geolocation searches.

BooksRepository

our repository inherits from the repository hierarchy and adds additional methods that are automatically implemented by Spring Data MongoDB

@Document is not Technically Required



Technically, the `@Document` annotation is not required unless mapping to a non-default collection. However, `@Document` will continue to be referenced in this lecture to mean the "subject of the repository".

Chapter 356. BooksRepository

All we need to create a functional repository is a `@Document` class and a primary key type. The `@Document` annotation is optional and only required to specify a collection name different from the class name. From our work to date, we know that our `@Document` is the Book class and the primary key is the primitive `String` type. This type works well with MongoDB auto-generated IDs.

356.1. Book @Document

Book @Document Example

```
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;
...
@Document(collection = "books")
public class Book {
    @Id
    private String id;
```

Multiple @Id Annotations, Use Spring Data's @Id Annotation

The `@Id` annotation looks the same as the JPA `@Id`, but instead comes from the Spring Data package



```
import org.springframework.data.annotation.Id;
```

356.2. BooksRepository

We declare our repository to extend `MongoRepository`.

```
public interface BooksRepository extends MongoRepository<Book, String> {}① ②
```

① Book is the repository type

② String is used for the primary key type



Consider Using Non-Primitive Primary Key Types

You will find that Spring Data MongoDB works easier with nullable object types.

Chapter 357. Configuration

Assuming your repository classes are in a package below the class annotated with `@SpringBootApplication`—not much is else is needed. Adding `@EnableMongoRepositories` is necessary when working with more complex classpaths.

Typical MongoDB Repository Support Declaration

```
@SpringBootApplication  
@EnableMongoRepositories  
public class MongoDBBooksApp {
```

If your repository is not located in the default packages scanned, their packages can be scanned with configuration options to the `@EnableMongoRepositories` annotation.

Configuring Repository Package Scanning

```
@EnableMongoRepositories(basePackageClasses = {BooksRepository.class}) ① ②
```

- ① the Java class provided here is used to identify the base Java package
- ② where to scan for repository interfaces

357.1. Injection

With the repository interface declared and the Mongo repository support enabled, we can then successfully inject the repository into our application.

BooksRepository Injection

```
@Autowired  
private BooksRepository booksRepository;
```

Chapter 358. CrudRepository

Lets start looking at the capability of our repository—starting with the declared methods of the `CrudRepository` interface.

`CrudRepository<T, ID> Interface`

```
public interface CrudRepository<T, ID> extends Repository<T, ID> {  
    <S extends T> S save(S);  
    <S extends T> Iterable<S> saveAll(Iterable<S>);  
    Optional<T> findById(ID);  
    boolean existsById(ID);  
    Iterable<T> findAll();  
    Iterable<T> findAllById(Iterable<ID>);  
    long count();  
    void deleteById(ID);  
    void delete(T);  
    void deleteAllById(Iterable<? extends ID>);  
    void deleteAll(Iterable<? extends T>);  
    void deleteAll();  
}  
  
public interface ListCrudRepository<T, ID> extends CrudRepository<T, ID> {  
    <S extends T> List<S> saveAll(Iterable<S>);  
    List<T> findAll();  
    List<T> findAllById(Iterable<ID>);  
}
```

358.1. CrudRepository save() New

We can use the `CrudRepository.save()` method to either create or update our `@Document` instance in the database. It has a direct correlation to MongoTemplate's `save()` method so there is not much extra functionality added by the repository layer.

In this specific example, we call `save()` with an object with an unassigned primary key. The primary key will be generated by the database when inserted and assigned to the object by the time the command completes.

`CrudRepository.save() New Example`

```
//given a transient document instance  
Book book = ...  
assertThat(book.getId()).isNull(); ①  
//when persisting  
booksRepo.save(book);  
//then document is persisted  
then(book.getId()).isNotNull(); ②
```

① document not yet assigned a generated primary key

② primary key assigned by database

358.2. CrudRepository save() Update Existing

The `CrudRepository.save()` method is an "upsert" method.

- if the `@Document` is new it will be inserted
- if a `@Document` exists with the currently assigned primary key, the original contents will be replaced

CrudRepository.save() Update Existing Example

```
//given a persisted document instance
Book book = ...
booksRepo.save(book); ①
Book updatedBook = book.withTitle("new title"); ②
//when persisting update
booksRepo.save(updatedBook);
//then new document state is persisted
then(booksRepo.findOne(Example.of(updatedBook))).isPresent(); ③
```

① object inserted into database — resulting in primary key assigned

② a separate instance with the same ID has modified title

③ object's new state is found in database

358.3. CrudRepository save()/Update Resulting MongoDB Command

Watching the low-level MongoDB commands, we can see that Mongo's built-in `upsert` capability allows the client to perform the action without a separate query.

MongoDB Update Command Performed with Upsert

```
update{"q":{"_id":{"$oid":"606cbfc0932e084392422bb6"}}, ①
  "u":{"_id":{"$oid":"606cbfc0932e084392422bb6"},"title":"new title","author":...},
  "multi":false,
  "upsert":true} ②
```

① filter looks for ID

② insert if not exist, update if exists

358.4. CrudRepository existsById()

The repository adds a convenience method that can check whether the `@Document` exists in the database without already having an instance or writing a criteria query.

The following snippet demonstrates how we can check for the existence of a given ID.

CrudRepository existsById()

```
//given a persisted document instance
Book pojoBook = ...
booksRepo.save(pojoBook);
//when - determining if document exists
boolean exists = booksRepo.existsById(pojoBook.getId());
//then
then(exists).isTrue();
```

The resulting MongoDB command issued a query for the ID, limiting the results to a single result, and a projection with only the primary key contained.

CrudRepository existsById() SQL

```
query: { _id: ObjectId('606cc5d742931870e951e08e') }
sort: {}
projection: {} ①
collation: { locale: \"simple\" }
limit: 1
```

① `projection: {}` returns only the primary key

358.5. CrudRepository findById()

If we need the full object, we can always invoke the `findById()` method, which should be a thin wrapper above `MongoTemplate.find()`, except that the return type is a Java `Optional<T>` versus the `@Document` type (`T`).

CrudRepository.findById()

```
//given a persisted document instance
Book pojoBook = ...
booksRepo.save(pojoBook);
//when - finding the existing document
Optional<Book> result = booksRepo.findById(pojoBook.getId()); ①
//then
then(result.isPresent()).isTrue();
```

① `findById()` always returns a non-null `Optional<T>` object

358.5.1. CrudRepository findById() Found Example

The `Optional<T>` can be safely tested for existence using `isPresent()`. If `isPresent()` returns `true`, then `get()` can be called to obtain the targeted `@Document`.

Present Optional Example

```
//given
then(result).isPresent();
//when - obtaining the instance
Book dbBook = result.get();
//then - instance provided
then(dbBook).isNotNull();
//then - database copy matches initial POJO
then(dbBook.getAuthor()).isEqualTo(pojoBook.getAuthor());
then(dbBook.getTitle()).isEqualTo(pojoBook.getTitle());
then(pojoBook.getPublished()).isEqualTo(dbBook.getPublished());
```

358.5.2. CrudRepository findById() Not Found Example

If `isPresent()` returns `false`, then `get()` will throw a `NoSuchElementException` if called. This gives your code some flexibility for how you wish to handle a target `@Document` not being found.

Missing Optional Example

```
//then - the optional can be benignly tested
then(result).isNotPresent();
//then - the optional is asserted during the get()
assertThatThrownBy(() -> result.get())
    .isInstanceOf(NoSuchElementException.class);
```

358.6. CrudRepository delete()

The repository also offers a wrapper around `MongoTemplate.remove()` that accepts an instance. Whether the instance existed or not, a successful call will always result in the `@Document` no longer in the database.

CrudRepository delete() Example

```
//when - deleting an existing instance
booksRepo.delete(existingBook);
//then - instance will be removed from DB
then(booksRepo.existsById(existingBook.getId())).isFalse();
```

358.6.1. CrudRepository delete() Not Exist

If the instance did not exist, the `delete()` call silently returns.

CrudRepository delete() Does Not Exists Example

```
//when - deleting a non-existing instance
booksRepo.delete(doesNotExist);
```

358.7. CrudRepository deleteById()

The repository also offers a convenience `deleteById()` method taking only the primary key.

CrudRepository deleteById() Example

```
//when - deleting an existing instance  
booksRepo.deleteById(existingBook.getId());
```

358.8. Other CrudRepository Methods

That was a quick tour of the `CrudRepository<T, ID>` interface methods. The following snippet shows the methods not covered. Most provide convenience methods around the entire repository.

Other CrudRepository Methods

```
//public interface CrudRepository<T, ID> extends Repository<T, ID> {  
    <S extends T> Iterable<S> saveAll(Iterable<S>);  
    Iterable<T> findAll();  
    Iterable<T> findAllById(Iterable<ID>);  
    long count();  
    void deleteAll(Iterable<? extends T>);  
    void deleteAll();  
  
//public interface ListCrudRepository<T, ID> extends CrudRepository<T, ID> {  
    <S extends T> List<S> saveAll(Iterable<S>);  
    List<T> findAll();  
    List<T> findAllById(Iterable<ID>);
```

Chapter 359. PagingAndSortingRepository

Before we get too deep into queries, it is good to know that Spring Data MongoDB has first-class support for sorting and paging.

- **sorting** - determines the order which matching results are returned
- **paging** - breaks up results into chunks that are easier to handle than entire database collections

Here is a look at the declared methods of the `PagingAndSortingRepository<T, ID>` interface. This defines extra parameters for the `CrudRepository.findAll()` methods.

PagingAndSortingRepository<T, ID> Interface

```
public interface PagingAndSortingRepository<T, ID> extends CrudRepository<T, ID> {  
    Iterable<T> findAll(Sort var1);  
    Page<T> findAll(Pageable var1);  
}
```

We will see paging and sorting option come up in many other query types as well.



Use Paging and Sorting for Collection Queries

All queries that return a collection should seriously consider adding paging and sorting parameters. Small test databases can become significantly populated production databases over time and cause eventual failure if paging and sorting is not applied to unbounded collection query return methods.

359.1. Sorting

Sorting can be performed on one or more properties and in ascending and/or descending order.

The following snippet shows an example of calling the `findAll()` method and having it return

- `Book` entities in descending order according to `published` date
- `Book` entities in ascending order according to `id` value when `published` dates are equal

Sort.by() Example

```
//when  
List<Book> byPublished = booksRepository.findAll(  
    Sort.by("published").descending().and(Sort.by("id").ascending()));① ②  
//then  
LocalDate previous = null;  
for (Book s: byPublished) {  
    if (previous!=null) {  
        then(previous).isAfterOrEqualTo(s.getPublished()); //DESC order  
    }  
    previous=s.getPublished();
```

```
}
```

- ① results can be sorted by one or more properties
- ② order of sorting can be ascending or descending

The following snippet shows how the MongoDB command was impacted by the `Sort.by()` parameter.

Sort.by() Example MongoDB Command

```
query: {}
sort: { published: -1, _id: 1 } ①
projection: {}
```

- ① `Sort.by()` added the extra sort parameters to MongoDB command

359.2. Paging

Paging permits the caller to designate how many instances are to be returned in a call and the offset to start that group (called a page or slice) of instances.

The snippet below shows an example of using one of the factory methods of `Pageable` to create a `PageRequest` definition using page size (limit), offset, and sorting criteria. If many pages will be traversed—it is advised to sort by a property that will produce a stable sort over time during table modifications.

Defining Initial Pageable

```
//given
int offset = 0;
int pageSize = 3;
Pageable pageable = PageRequest.of(offset/pageSize, pageSize, Sort.by("published"));①
//when
Page<Book> bookPage = booksRepository.findAll(pageable);
```

- ① using `PageRequest` factory method to create `Pageable` from provided page information

Use Stable Sort over Large Collections



Try to use a property for sort (at least by default) that will produce a stable sort when paging through a large collection to avoid repeated or missing objects from follow-on pages because of new changes to the table.

359.3. Page Result

The page result is represented by a container object of type `Page<T>`, which extends `Slice<T>`. I will describe the difference next, but the `PagingAndSortingRepository<T, ID>` interface always returns a `Page<T>`, which will provide:

- the sequential number of the page/slice
- the requested size of the page/slice
- the number of elements found
- the total number of elements available in the database

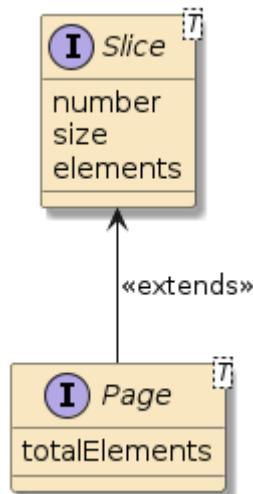


Figure 154. *Page<T>* Extends *Slice<T>*

359.4. Slice Properties

The *Slice<T>* base interface represents properties about the content returned.

Slice Properties

```
//then
Slice bookSlice = bookPage; ①
then(bookSlice).isNotNull();
then(bookSlice.isEmpty()).toBeFalsy();
then(bookSlice.getNumber()).isEqualTo(0); ②
then(bookSlice.getSize()).isEqualTo(pageSize); ③
then(bookSlice.getNumberOfElements()).isEqualTo(pageSize); ④

List<Book> booksList = bookSlice.getContent();
then(booksList).hasSize(pageSize);
```

① *Page<T>* extends *Slice<T>*

② slice increment — first slice is 0

③ the number of elements requested for this slice

④ the number of elements returned in this slice

359.5. Page Properties

The *Page<T>* derived interface represents properties about the entire collection/table.

The snippet below shows an example of the total number of elements in the table being made available to the caller.

Page Properties

```
then(bookPage.getTotalElements()).isEqualTo(savedBooks.size());
```

359.6. Stateful Pageable Creation

In the above example, we created a `Pageable` from stateless parameters. We can also use the original `Pageable` to generate the next or other relative page specifications.

Relative Pageable Creation

```
Pageable pageable = PageRequest.of(offset / pageSize, pageSize, Sort.by("published"));
...
Pageable next = pageable.next();
Pageable previous = pageable.previousOrFirst();
Pageable first = pageable.first();
```

359.7. Page Iteration

The next `Pageable` can be used to advance through the complete set of query results, using the previous `Pageable` and testing the returned `Slice`.

Page Iteration

```
for (int i=1; bookSlice.hasNext(); i++) { ①
    pageable = pageable.next(); ②
    bookSlice = booksRepository.findAll(pageable);
    booksList = bookSlice.getContent();
    then(bookSlice).isNotNull();
    then(bookSlice.getNumber()).isEqualTo(i);
    then(bookSlice.getSize()).isEqualTo(pageSize);
    then(bookSlice.getNumberOfElements()).isLessThanOrEqualTo(pageSize);
    then(((Page)bookSlice).getTotalElements()).isEqualTo(savedBooks.size());//unique
    to Page
}
then(bookSlice.hasNext()).toBeFalsy();
then(bookSlice.getNumber()).isEqualTo(booksRepository.count() / pageSize);
```

① `Slice.hasNext()` will indicate when previous `Slice` represented the end of the results

② next `Pageable` obtained from previous `Pageable`

Chapter 360. Query By Example

Not all queries will be as simple as `findAll()`. We now need to start looking at queries that can return a subset of results based on them matching a set of predicates. The `QueryByExampleExecutor<T>` parent interface to `MongoRepository<T, ID>` provides a set of variants to the collection-based results that accepts an "example" to base a set of predicates off of.

QueryByExampleExecutor<T> Interface

```
public interface QueryByExampleExecutor<T> {  
    <S extends T> Optional<S> findOne(Example<S>);  
    <S extends T> Iterable<S> findAll(Example<S>);  
    <S extends T> Iterable<S> findAll(Example<S>, Sort);  
    <S extends T> Page<S> findAll(Example<S>, Pageable);  
    <S extends T> long count(Example<S>);  
    <S extends T> boolean exists(Example<S>);  
    <S extends T, R> R findBy(Example<S>, Function<FluentQuery$FetchableFluentQuery<S>, R>);  
}
```

360.1. Example Object

An `Example` is an interface with the ability to hold onto a probe and matcher.

360.1.1. Probe Object

The probe is an instance of the repository `@Document` type.

The following snippet is an example of creating a probe that represents the fields we are looking to match.

Probe Example

```
//given  
Book savedBook = savedBooks.get(0);  
Book probe = Book.builder()  
    .title(savedBook.getTitle())  
    .author(savedBook.getAuthor())  
    .build(); ①
```

① probe will carry values for `title` and `author` to match

360.1.2. ExampleMatcher Object

The matcher defaults to an exact match of all non-null properties in the probe. There are many definitions we can supply to customize the matcher.

- `ExampleMatcher.matchingAny()` - forms an OR relationship between all predicates

- `ExampleMatcher.matchingAll()` - forms an AND relationship between all predicates

The `matcher` can be broken down into specific fields, designing a fair number of options for String-based predicates but very limited options for non-String fields.

- exact match
- case insensitive match
- starts with, ends with
- contains
- regular expression
- include or ignore nulls

The following snippet shows an example of the default `ExampleMatcher`.

Default ExampleMatcher

```
ExampleMatcher matcher = ExampleMatcher.matching(); ①
```

① default matcher is `matchingAll`

360.2. findAll By Example

We can supply an `Example` instance to the `findAll()` method to conduct our query.

The following snippet shows an example of using a probe with a default matcher. It is intended to locate all books matching the `author` and `title` we specified in the probe.

```
//when
List<Book> foundBooks = booksRepository.findAll(
    Example.of(probe), //default matcher is matchingAll() and non-null
    Sort.by("id"));
```

The default matcher ends up working perfectly with our `@Document` class because a nullable primary key was used—keeping the primary key from being added to the criteria.

360.3. Ignoring Properties

If we encounter any built-in types that cannot be null—we can configure a match to explicitly ignore certain fields.

The following snippet shows an example matcher configured to ignore the primary key.

matchingAll ExampleMatcher with Ignored Property

```
ExampleMatcher ignoreId = ExampleMatcher.matchingAll().withIgnorePaths("id"); ①
//when
List<Book> foundBooks = booksRepository.findAll(
    Example.of(probe, ignoreId), ②
    Sort.by("id"));
//then
then(foundBooks).isNotEmpty();
```

```
then(foundBooks.get(0).getId()).isEqualTo(savedBook.getId());
```

- ① **id** primary key is being excluded from predicates
- ② non-null and non-id fields of probe are used for **AND** matching

360.4. Contains ExampleMatcher

We have some options on what we can do with the String matches.

The following snippet provides an example of testing whether **title** contains the text in the probe while performing an exact match of the **author** and ignoring the **id** field.

Contains ExampleMatcher

```
Book probe = Book.builder()  
    .title(savedBook.getTitle().substring(2))  
    .author(savedBook.getArtist())  
    .build();  
ExampleMatcher matcher = ExampleMatcher  
    .matching()  
    .withIgnorePaths("id")  
    .withMatcher("title", ExampleMatcher.GenericPropertyMatchers.contains());
```

360.4.1. Using Contains ExampleMatcher

The following snippet shows that the **Example** successfully matched on the **Book** we were interested in.

Example is Found

```
//when  
List<Book> foundBooks=booksRepository.findAll(Example.of(probe,matcher), Sort.by("id"));  
//then  
then(foundBooks).isNotEmpty();  
then(foundBooks.get(0).getId()).isEqualTo(savedBook.getId());
```

Chapter 361. Derived Queries

For fairly straight forward queries, Spring Data MongoDB can derive the required commands from a method signature declared in the repository interface. This provides a more self-documenting version of similar queries we could have formed with query-by-example.

The following snippet shows a few example queries added to our repository interface to address specific queries needed in our application.

Example Query Method Names

```
public interface BooksRepository extends MongoRepository<Book, String> {  
    Optional<Book> getByTitle(String title); ①  
  
    List<Book> findByTitleNullAndPublishedAfter(LocalDate date); ②  
  
    List<Book> findByTitleStartingWith(String string, Sort sort); ③  
    Slice<Book> findByTitleStartingWith(String string, Pageable pageable); ④  
    Page<Book> findPageByTitleStartingWith(String string, Pageable pageable); ⑤
```

- ① query by an exact match of `title`
- ② query by a match of two fields (`title` and `released`)
- ③ query using sort
- ④ query with paging support
- ⑤ query with paging support and table total

Let's look at a complete example first.

361.1. Single Field Exact Match Example

In the following example, we have created a query method `getByTitle` that accepts the exact match title value and an `Optional` return value.

Interface Method Signature

```
Optional<Book> getByTitle(String title); ①
```

We use the declared interface method in a normal manner and Spring Data MongoDB takes care of the implementation.

Interface Method Usage

```
//when  
Optional<Book> result = booksRepository.getByTitle(book.getTitle());  
//then  
then(result.isPresent()).isTrue();
```

The result is essentially the same as if we implemented it using query-by-example or more directly through the MongoTemplate.

361.2. Query Keywords

Spring Data has several **keywords**, followed by **By**, that it looks for starting the interface method name. Those with multiple terms can be used interchangeably.

Meaning	Keywords
Query	<ul style="list-style-type: none">• find• read• get• query• search• stream
Count	<ul style="list-style-type: none">• count
Exists	<ul style="list-style-type: none">• exists
Delete	<ul style="list-style-type: none">• delete• remove

361.3. Other Keywords

Other keywords are clearly documented in the JPA reference [\[1\]](#) [\[2\]](#)

- Distinct (e.g., `findDistinctByTitle`)
- Is, Equals (e.g., `findByTitle`, `findByTitleIs`, `findByTitleEquals`)
- Not (e.g., `findByTitleNot`, `findByTitleIsNot`, `findByTitleNotEquals`)
- IsNull, IsNotNull (e.g., `findByTitle(null)`, `findByTitleIsNull()`, `findByTitleIsNotNull()`)
- StartingWith, EndingWith, Containing (e.g., `findByTitleStartingWith`, `findByTitleEndingWith`, `findByTitleContaining`)
- LessThan, LessThanEqual, GreaterThan, GreaterThanEqual, Between (e.g., `findByIdLessThan`, `findByIdBetween(lo,hi)`)
- Before, After (e.g., `findByPublishedAfter`)
- In (e.g., `findByTitleIn(collection)`)
- OrderBy (e.g., `findByTitleContainingOrderByTitle`)

The list is significant, but not meant to be exhaustive. Perform a web search for your specific needs (e.g., "Spring Data Derived Query ...") if what is needed is not found here.

361.4. Multiple Fields

We can define queries using one or more fields using **And** and **Or**.

The following example defines an interface method that will test two fields: `title` and `published`. `title` will be tested for null and `published` must be after a certain date.

Multiple Fields Interface Method Declaration

```
List<Book> findByTitleNullAndPublishedAfter(LocalDate date);
```

The following snippet shows an example of how we can call/use the repository method. We are using a simple collection return without sorting or paging.

Multiple Fields Example Use

```
//when
List<Book> foundBooks = booksRepository.findByTitleNullAndPublishedAfter(firstBook
    .getPublished());
//then
Set<String> foundIds = foundBooks.stream().map(s->s.getId()).collect(Collectors.toSet
());
then(foundIds).isEqualTo(expectedIds);
```

361.5. Collection Response Query Example

We can perform queries with various types of additional arguments and return types. The following shows an example of a query that accepts a sorting order and returns a simple collection with all objects found.

Collection Response Interface Method Declaration

```
List<Book> findByTitleStartingWith(String string, Sort sort);
```

The following snippet shows an example of how to form the `Sort` and call the query method derived from our interface declaration.

Collection Response Interface Method Use

```
//when
Sort sort = Sort.by("id").ascending();
List<Book> books = booksRepository.findByTitleStartingWith(startingWith, sort);
//then
then(books.size()).isEqualTo(expectedCount);
```

361.6. Slice Response Query Example

Derived queries can also be declared to accept a `Pageable` definition and return a `Slice`. The following example shows a similar interface method declaration to what we had prior — except we have wrapped the `Sort` within a `Pageable` and requested a `Slice`, which will contain only those items that match the predicate and comply with the paging constraints.

Slice Response Interface Method Declaration

```
Slice<Book> findByTitleStartingWith(String string, Pageable pageable);
```

The following snippet shows an example of forming the `PageRequest`, making the call, and inspecting the returned `Slice`.

Slice Response Interface Method Use

```
//when
PageRequest pageable=PageRequest.of(0, 1, Sort.by("id").ascending());① ②
Slice<Book> booksSlice=booksRepository.findByTitleStartingWith(startingWith,pageable);
//then
then(booksSlice.getNumberElements()).isEqualTo(pageable.getPageSize());
```

① `pageNumber` is 0

② `pageSize` is 1

361.7. Page Response Query Example

We can alternatively declare a `Page` return type if we also need to know information about all available matches in the table. The following shows an example of returning a `Page`. The only reason `Page` shows up in the method name is to form a different method signature than its sibling examples. `Page` is not required to be in the method name.

Page Response Interface Method Declaration

```
Page<Book> findPageByTitleStartingWith(String string, Pageable pageable); ①
```

① the `Page` return type (versus `Slice`) triggers an extra query performed to supply `totalElements` `Page` property

The following snippet shows how we can form a `PageRequest` to pass to the derived query method and accept a `Page` in response with additional table information.

Page Response Interface Method Use

```
//when
PageRequest pageable = PageRequest.of(0, 1, Sort.by("id").ascending());
Page<Book> booksPage = booksRepository.findPageByTitleStartingWith(startingWith,
pageable);
//then
then(booksPage.getNumberElements()).isEqualTo(pageable.getPageSize());
then(booksPage.getTotalElements()).isEqualTo(expectedCount); ①
```

① an extra property is available to tell us the total number of matches relative to the entire table — that may not have been reported on the current page

[1] "Query Creation", Spring Data JPA - Reference Documentation

Chapter 362. @Query Annotation Queries

Spring Data MongoDB provides an option for the query to be expressed on the repository method.

The following example will locate a book published between the provided dates—inclusive. The default derived query implemented it exclusive of the two dates. The `@Query` annotation takes precedence over the default derived query. This shows how easy it is to define a customized version of the query.

Example @Query

```
@Query("{ 'published': { $gte: ?0, $lte: ?1 } }") ①
List<Book> findByPublishedBetween(LocalDate starting, LocalDate ending);
```

① `?0` is the first parameter (`starting`) and `?1` is the second parameter (`ending`)

The following snippet shows an example of implementing a query using a regular expression completed by the input parameters. It locates all books with `titles` greater-than or equal to the `length` parameter. It also declares that only the `title` field of the `Book` instances need to be returned—making the result smaller.

Query Supplied on Repository Method

```
@Query(value="{ 'title': /^.{?0,}$/ }", fields="{'_id':0, 'title':1}") ① ②
List<Book> getTitlesGESizeAsBook(int length);
```

① `value` expresses which Books should match

② `fields` expresses which fields should be returned and populated in the instance

Named Queries can be supplied in property file

Named queries can also be expressed in a property file—versus being placed directly onto the method. Property files can provide a more convenient source for expressing more complex queries.



```
@EnableMongoRepositories(namedQueriesLocation="...")
```

The default location is `META-INF/mongo-named-queries.properties`

362.1. @Query Annotation Attributes

The matches in the query can be used for more than just `find`. We can alternately apply `count`, `exists`, or `delete` and include information for `fields` projected, `sort`, and `collation`.

Table 25. @Query Annotation Attributes

Attribute	Default	Description	Example
String fields	""	projected fields	fields = "{ title : 1 }"
boolean count	false	count() action performed on query matches	
boolean exists	false	exists() action performed on query matches	
boolean delete	false	delete() action performed on query matches	
String sort	""	sort expression for query results	sort = "{ published : -1 }"
String collation	""	location information	

362.2. @Query Sort and Paging

The `@Query` approach supports paging via `Pageable` parameter. Sort must be defined using the `@Query.sort` property.

@Query Sort and Paging

```

@Query
@Query(value="{ 'published': { $gte: ?0, $lte: ?1 } }", sort = "{ '_id':1 }")
Page<Book> findByPublishedBetween(LocalDate starting, LocalDate ending, Pageable
pageable);

```

Chapter 363. MongoRepository Methods

Many of the methods and capabilities of the `MongoRepository<T, ID>` are available at the higher level interfaces. The `MongoRepository<T, ID>` itself declares two types of additional methods

- insert/upsert state-specific optimizations
- return type extensions

MongoRepository<T, ID> Interface

```
<S extends T> S insert(S); ①
<S extends T> List<S> insert(Iterable<S>);

<S extends T> List<S> findAll(Example<S>); ②
<S extends T> List<S> findAll(Example<S>, Sort);
public default Iterable findAll(Example, Sort);
public default Iterable findAll(Example);
```

① `insert` is specific to `MongoRepository` and assumes the document is new

② `List<T>` is a sub-type of `Iterable<T>` and provides a richer set of inspection methods for the returned result from `QueryByExample` methods

Chapter 364. Custom Queries

Sooner or later, a repository action requires some complexity that is beyond the ability to leverage a single query-by-example or derived query. We may need to implement some custom logic or may want to encapsulate multiple calls within a single method.

364.1. Custom Query Interface

The following example shows how we can extend the repository interface to implement custom calls using the MongoTemplate and the other repository methods. Our custom implementation will return a random `Book` from the database.

Interface for Public Custom Query Methods

```
public interface BooksRepositoryCustom {  
    Optional<Book> random();  
}
```

364.2. Repository Extends Custom Query Interface

We then declare the repository to extend the additional custom query interface—making the new method(s) available to callers of the repository.

Repository Implements Custom Query Interface

```
public interface BooksRepository extends MongoRepository<Book, String>,  
BooksRepositoryCustom { ①  
    ...
```

① added additional `BookRepositoryCustom` interface for `BookRepository` to extend

364.3. Custom Query Method Implementation

Of course, the new interface will need an implementation. This will require at least two lower-level database calls

1. determine how many objects there are in the database
2. return an instance for one of those random values

The following snippet shows a portion of the custom method implementation. Note that two additional helper methods are required. We will address them in a moment. By default, this class must have the same name as the interface, followed by "Impl".

Custom Query Method Implementation

```
public class BookRepositoryCustomImpl implements BookRepositoryCustom {  
    private final SecureRandom random = new SecureRandom();
```

```

...
@Override
public Optional<Book> random() {
    Optional randomBook = Optional.empty();
    int count = (int) booksRepository.count(); ①

    if (count!=0) {
        int offset = random.nextInt(count);
        List<Book> books = books(offset, ②
            randomBook=books.isEmpty() ? Optional.empty() : Optional.of(books.get(0));
    }
    return randomBook;
}

```

① leverages `CrudRepository.count()` helper method

② leverages a local, private helper method to access specific `Book`

364.4. Repository Implementation Postfix

If you have an alternate suffix pattern other than "Impl" in your application, you can set that value in an attribute of the `@EnableMongoRepositories` annotation.

The following shows a declaration that sets the suffix to its normal default value (i.e., we did not have to do this). If we changed this value from "Impl" to "Xxx", then we would need to change `BooksRepositoryCustomImpl` to `BooksRepositoryCustomXxx`.

Optional Custom Query Method Implementation Suffix

```
@EnableMongoRepositories(repositoryImplementationPostfix="Impl")①
```

① `Impl` is the default value. Configure this attribute to use non-`Impl` postfix

364.5. Helper Methods

The custom `random()` method makes use of two helper methods. One is in the `CrudRepository` interface and the other directly uses the `MongoTemplate` to issue a query.

CrudRepository.count() Used as Helper Method

```
public interface CrudRepository<T, ID> extends Repository<T, ID> {
    long count();
```

EntityManager NamedQuery used as Helper Method

```
protected List<Book> books(int offset, int limit) {
    return mongoTemplate.find(new Query().skip(offset).limit(limit), Book.class);
}
```

We will need to inject some additional resources in order to make these calls:

- BooksRepository
- MongoTemplate

364.6. Naive Injections

Since we are not using sessions or transactions with Mongo, a simple/naive injection will work fine. We do not have to worry about injecting a specific instance. However, we will run into a circular dependency issue with the BooksRepository.

Naive Injections

```
@RequiredArgsConstructor
public class BooksRepositoryCustomImpl implements BooksRepositoryCustom {
    private final MongoTemplate mongoTemplate; ①
    private final BooksRepository booksRepository; ②
```

- ① any MongoTemplate instance referencing the correct database and collection is fine
② eager/mandatory injection of self needs to be delayed

364.7. Required Injections

We need to instead

- use `@Autowired @Lazy` and a non-final attribute for the BooksRepository injection to indicate that this instance can be initialized without access to the injected bean

Required Injections

```
import org.springframework.data.jpa.repository.MongoContext;
...
public class BooksRepositoryCustomImpl implements BooksRepositoryCustom {
    private final MongoTemplate mongoTemplate;
    @Autowired @Lazy ①
    private BooksRepository booksRepository;
```

- ① BooksRepository lazily injected to mitigate the recursive dependency between the `Impl` class and the full repository instance

364.8. Calling Custom Query

With all that in place, we can then call our custom `random()` method and obtain a sample Book to work with from the database.

Example Custom Query Client Call

```
//when
```

```
Optional<Book> randomBook = booksRepository.random();
//then
then(randomBook.isPresent()).isTrue();
```

364.9. Implementing Aggregation

MongoTemplate has more power in it than what can be expressed with MongoRepository. As seen with the `random()` implementation, we have the option of combining operations and dropping down the to `MongoTemplate` for a portion of the implementation. That can also include use of the Aggregation Pipeline, GridFS, Geolocation, etc.

The following custom implementation is declared in the Custom interface, extended by the BooksRepository.

Custom Query Interface Definition

```
public interface BookRepositoryCustom {
...
List<Book> findByAuthorGESize(int length);
```

The snippet below shows the example leveraging the Aggregation Pipeline for its implementation and returning a normal `List<Book>` collection.

Custom Query Implementation Based On Aggregation Pipeline

```
@Override
public List<Book> findByAuthorGESize(int length) {
    String expression = String.format("^.{%-d,}$$", length);

    Aggregation pipeline = Aggregation.newAggregation(
        Aggregation.match(Criteria.where("author").regex(expression)),
        Aggregation.match(Criteria.where("author").exists(true))
    );
    AggregationResults<Book> result =
        mongoTemplate.aggregate(pipeline, "books", Book.class);
    return result.getMappedResults();
}
```

That allows us unlimited behavior in the data access layer and the ability to encapsulate the capability into a single data access component.

Chapter 365. Summary

In this module, we learned:

- that Spring Data MongoDB eliminates the need to write boilerplate MongoTemplate code
- to perform basic CRUD management for `@Document` classes using a repository
- to implement query-by-example
- that unbounded collections can grow over time and cause our applications to eventually fail
 - that paging and sorting can easily be used with repositories
- to implement query methods derived from a query DSL
- to implement custom repository extensions

Mongo Repository End-to-End Application

copyright © 2024 jim stafford (jim.stafford@jhu.edu)

Chapter 366. Introduction

This lecture takes what you have learned in establishing a MongoDB data tier using Spring Data MongoDB and shows that integrated into an end-to-end application with API CRUD calls and finder calls using paging. It is assumed that you already know about API topics like Data Transfer Objects (DTOs), JSON and XML content, marshalling/unmarshalling using Jackson and JAXB, web APIs/controllers, and clients. This lecture will put them all together.



Due to the common component technologies between the Spring Data JPA and Spring Data MongoDB end-to-end solution, this lecture is about 95% the same as the Spring Data JPA End-to-End Application lecture. Although it is presumed that the Spring Data JPA End-to-End Application lecture precedes this lecture—it was written so that was not a requirement. If you have already mastered the Spring Data JPA End-to-End Application topics, you should be able to quickly breeze through this material because of the significant similarities in concepts and APIs.

366.1. Goals

The student will learn:

- to integrate a Spring Data MongoDB Repository into an end-to-end application, accessed through an API
- to make a clear distinction between Data Transfer Objects (DTOs) and Business Objects (BOs)
- to identify data type architectural decisions required for a multi-tiered application
- to understand the need for paging when working with potentially unbounded collections and remote clients

366.2. Objectives

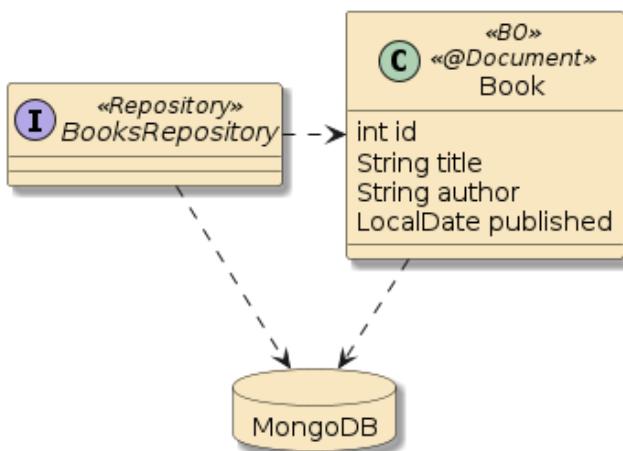
At the conclusion of this lecture and related exercises, the student will be able to:

1. implement a BO tier of classes that will be mapped to the database
2. implement a DTO tier of classes that will exchange state with external clients
3. implement a service tier that completes useful actions
4. identify the controller/service layer interface decisions when it comes to using DTO and BO classes
5. implement a mapping tier between BO and DTO objects
6. implement paging requests through the API
7. implement page responses through the API

Chapter 367. BO/DTO Component Architecture

367.1. Business Object(s)/@Documents

For our Books application—I have kept the data model simple and kept it limited to a single business object (BO) `@Document` class mapped to the database using Spring Data MongoDB annotations and accessed through a Spring Data MongoDB repository.



The business objects are the focal point of information where we implement our business decisions.

Figure 155. BO Class Mapped to DB as Spring Data MongoDB `@Document`

The primary focus of our BO classes is to map business implementation concepts to the database. There are two fundamental patterns of business objects:

- **Anemic Domain Model** - containing no validations, calculations, or implementation of business rules. A basic data mapping with getters and setters.
- **Rich Domain Model** - combining data with business behavior indicative of an Object-Oriented design. The rich domain model is at the heart of Domain Driven Design (DDD) architectural concepts.

Due to our simplistic business domain, the example business object is very anemic. Do not treat that as a desirable target for all cases.

The following snippet shows some of the optional mapping properties of a Spring Data MongoDB `@Document` class.

BO Class Sample Spring Data MongoDB Mappings

```
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;
import org.springframework.data.mongodb.core.mapping.Field;

@Document(collection = "books") ①
...
```

```

public class Book {
    @Id ②
    private String id;
    @Field(name="title") ③
    private String title;
    private String author;
    private LocalDate published;
    ...
}

```

① `@Document.collection` used to define the DB collection to use — otherwise uses name of class

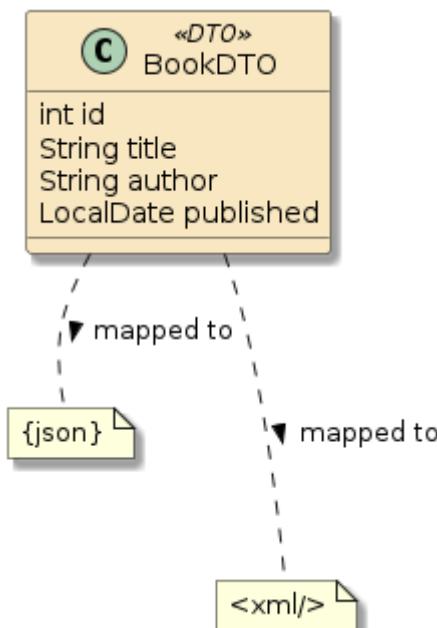
② `@Id` used to map the document primary key field to a class property

③ `@Field` used to custom map a class property to a document field — the example is performing what the default would have done

367.2. Data Transfer Object(s) (DTOs)

The Data Transfer Objects are the focal point of interfacing with external clients. They represent state at a point in time. For external web APIs, they are commonly mapped to both JSON and XML.

For the API, we have the decision of whether to reuse BO classes as DTOs or implement a separate set of classes for that purpose. Even though some applications start out simple, there will come a point where database technology or mappings will need to change at a different pace than API technology or mappings.



For that reason, I created a separate `BooksDTO` class to represent a sample DTO. It has a near 1:1 mapping with the `Book` BO. This 1:1 representation of information makes it seem like this is an unnecessary extra class, but it demonstrates an initial technical separation between the DTO and BO that allows for independent changes down the road.

Figure 156. DTO

The primary focus of our DTO classes is to map business interface concepts to a portable exchange format.

367.3. BookDTO Class

The following snippet shows some of the annotations required to map the `BookDTO` class to XML using Jackson and JAXB. Jackson JSON requires very few annotations in the simple cases.

DTO Class Sample JSON/XML Mappings

```
import com.fasterxml.jackson.dataformat.xml.annotation.JacksonXmlRootElement;
import com.fasterxml.jackson.dataformat.xml.annotation.JacksonXmlProperty;
import jakarta.xml.bind.annotation.XmlRootElement;
import jakarta.xml.bind.annotation.XmlAccessType;
import jakarta.xml.bind.annotation.XmlAccessorType;
...
@JacksonXmlRootElement(localName = "book", namespace = "urn:ejava.db-repo.books")
@XmlRootElement(name = "book", namespace = "urn:ejava.db-repo.books") ②
@XmlAccessorType(XmlAccessType.FIELD)
@NoArgsConstructor
...
public class BookDTO { ①
    @JacksonXmlProperty(isAttribute = true)
    @XmlAttribute
    private String id;
    private String title;
    private String author;
    @XmlJavaTypeAdapter(LocalDateJaxbAdapter.class) ③
    private LocalDate published;
...
}
```

① Jackson JSON requires very little to no annotations for simple mappings

② XML mappings require more detailed definition to be complete

③ JAXB requires a custom mapping definition for java.time types

367.4. BO/DTO Mapping

With separate BO and DTO classes, there is a need for mapping between the two.

- map from DTO to BO for requests
- map from BO to DTO for responses

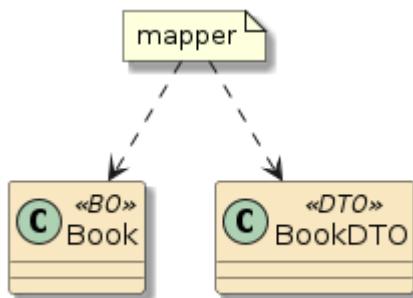


Figure 157. BO to DTO Mapping

We have several options on how to organize this role.

367.4.1. BO/DTO Self Mapping

- The BO or the DTO class can map to the other
 - Benefit: good encapsulation of detail within the data classes themselves
 - Drawback: promotes coupling between two layers we were trying to isolate



Avoid unless users of DTO will be tied to BO and are just exchanging information.

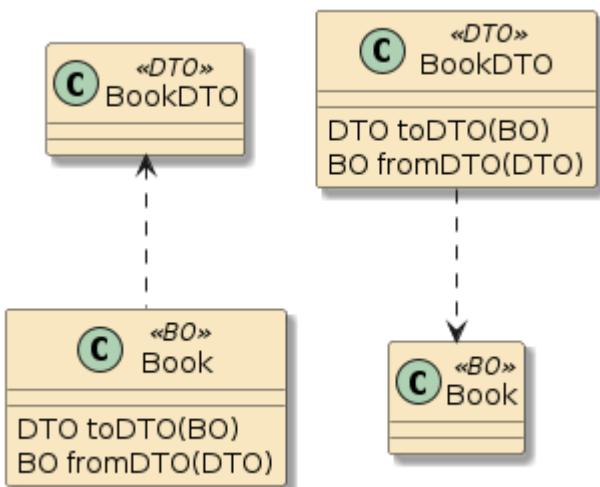


Figure 158. BO to DTO Self Mapping

367.4.2. BO/DTO Method Self Mapping

- The API or service methods can map things themselves within the body of the code
 - Benefit: mapping specialized to usecase involved
 - Drawback:
 - mixed concerns within methods.
 - likely have repeated mapping code in many methods



Avoid.

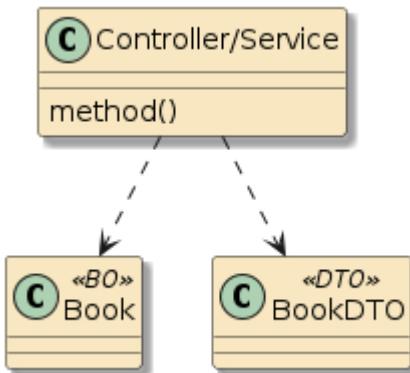


Figure 159. BO to DTO Method Self Mapping

367.4.3. BO/DTO Helper Method Mapping

- Delegate mapping to a reusable helper method within the API or service classes
 - Benefit: code reuse within the API or service class
 - Drawback: potential for repeated mapping in other classes



This is a small but significant step to a helper class

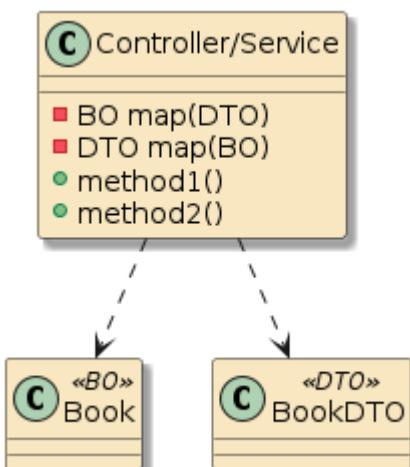


Figure 160. BO/DTO Helper Method Mapping

367.4.4. BO/DTO Helper Class Mapping

- Create a separate interface/class to inject into the API or service classes that encapsulates the role of mapping
 - Benefit: Reusable, testable, separation of concern
 - Drawback: none



Best in most cases unless good reason for self-mapping is appropriate.

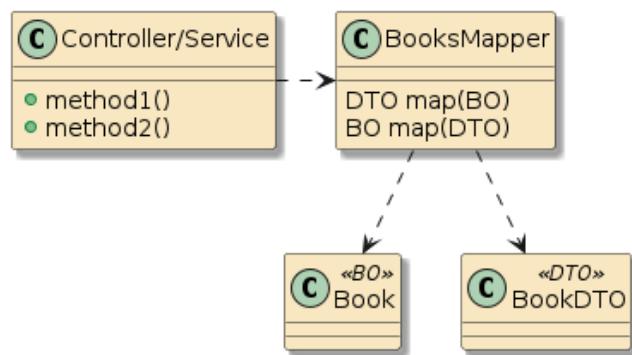


Figure 161. BO/DTO Helper Class Mapping

367.4.5. BO/DTO Helper Class Mapping Implementations

Mapping helper classes can be implemented by:

- brute force implementation
 - Benefit: likely the fastest performance and technically simplest to understand
 - Drawback: tedious setter/getter code
- off-the-shelf mapper libraries (e.g. [Dozer](#), [Orika](#), [MapStruct](#), [ModelMapper](#), [JMapper](#))^[1] ^[2]
 - Benefit:
 - declarative language and inferred DIY mapping options
 - some rely on code generation at compile time (similar in lifecycle to Lombok in some ways) with the ability to override and customize
 - Drawbacks:
 - some rely on reflection for mapping which add to overhead
 - non-trivial mappings can be complex to understand



MapStruct Thumbs Up

I have personally used Dozer in detail (years ago) and have recently been introduced to MapStruct. I really like MapStruct much better. It writes much of the same code you would have written in the brute force approach—without using reflection at runtime. You can define a mapper through interfaces and abstract classes—depending on how much you need to customize. You can also declare the mapper as a component to have helper components injected for use in mapping. In the end, you get a class with methods written in Java source that you can clearly see. Everything is understandable.

[1] "Performance of Java Mapping Frameworks", Baeldung

[2] "any tool for java object to object mapping?", Stack Overflow

Chapter 368. Service Architecture

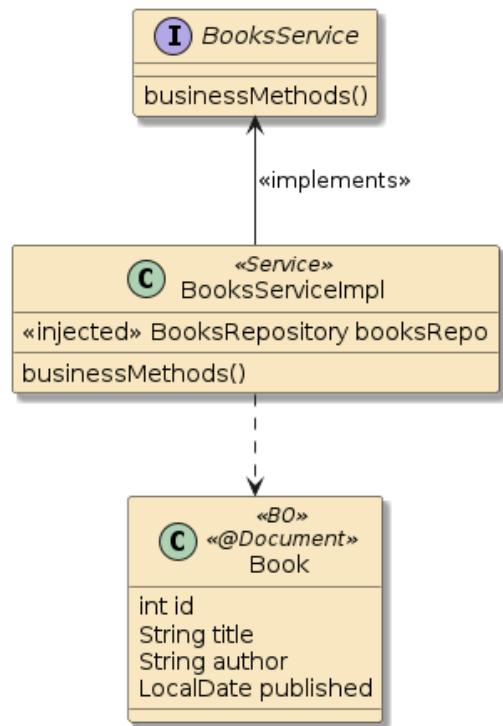
Services—with the aid of BOs—implement the meat of the business logic.

The service

- implements an interface with business methods
- is annotated with `@Service` component in most cases to self-support auto-injection
- injects repository component(s)
- interacts with BO instances

Example Service Class Declaration

```
@RequiredArgsConstructor
@Service
public class BooksServiceImpl
    implements BooksService {
    private final BooksMapper mapper;
    private final BooksRepository booksRepo;
    ...
}
```



368.1. Injected Service Boundaries

Container features like `@Secured`, `@Async`, etc. are only implemented at component boundaries. When a `@Component` dependency is injected, the container has the opportunity to add features using "interpose". As a part of interpose—the container implements proxy to add the desired feature of the target component method.

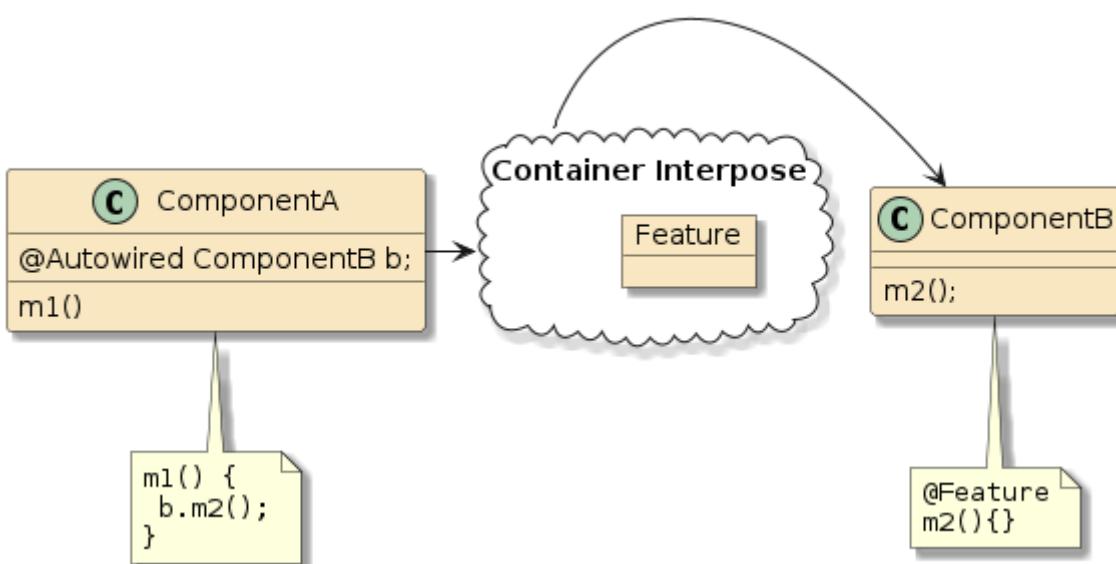


Figure 162. Container Interpose

Therefore it is important to arrange a component boundary wherever you need to start a new characteristic provided by the container. The following is a more detailed explanation of what not to do and do.

368.1.1. Buddy Method Boundary

The methods within a component class are not typically subject to container interpose. Therefore a call from m1() to m2() within the same component class is a straight Java call.



No Interpose for Buddy Method Calls

Buddy method calls are straight Java calls without container interpose.

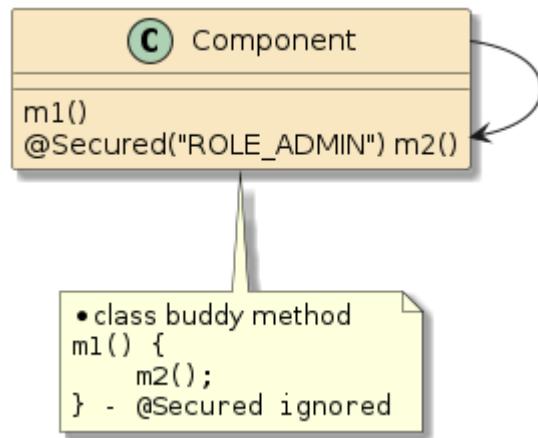


Figure 163. Buddy Method Boundary

368.1.2. Self Instantiated Method Boundary

Container interpose is only performed when the container has a chance to decorate the called component. Therefore, a call to a method of a component class that is self-instantiated will not have container interpose applied—no matter how the called method is annotated.



No Interpose for Self-Instantiated Components

Self-instantiated classes are not subject to container interpose.

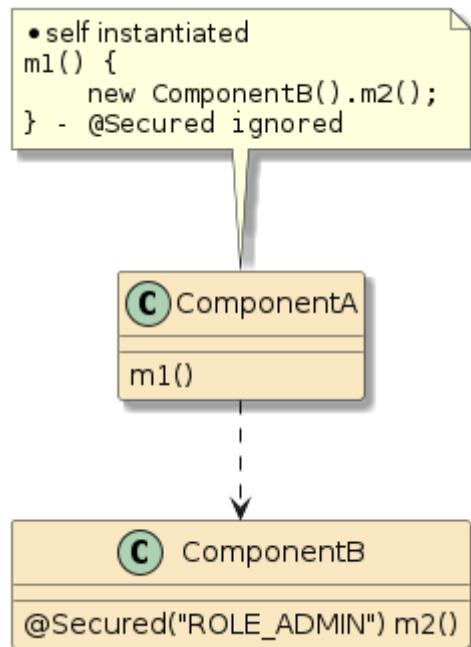


Figure 164. Self Instantiated Method Boundary

368.1.3. Container Injected Method Boundary

Components injected by the container are subject to container interpose and will have declared characteristics applied.



*Container-Injected Components
have Interpose*

Use container injection to have declared features applied to called component methods.

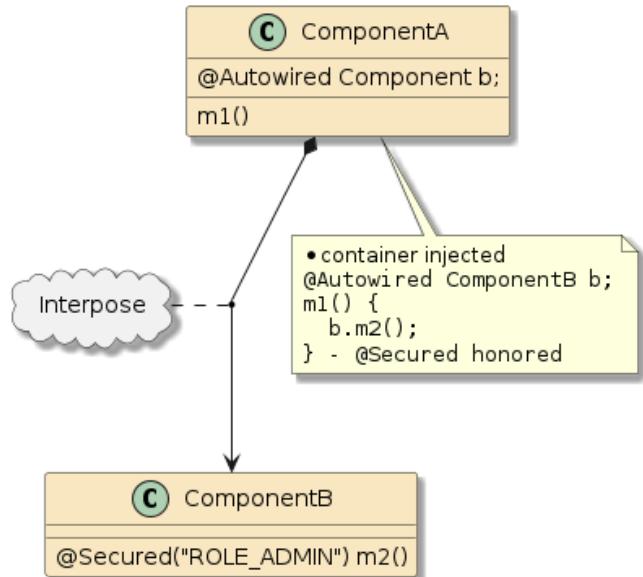


Figure 165. Container Injected Method Boundary

368.2. Compound Services

With `@Component` boundaries and interpose constraints understood—in more complex security, or threading solutions, the logical `@Service` many get broken up into one or more physical helper `@Component` classes.

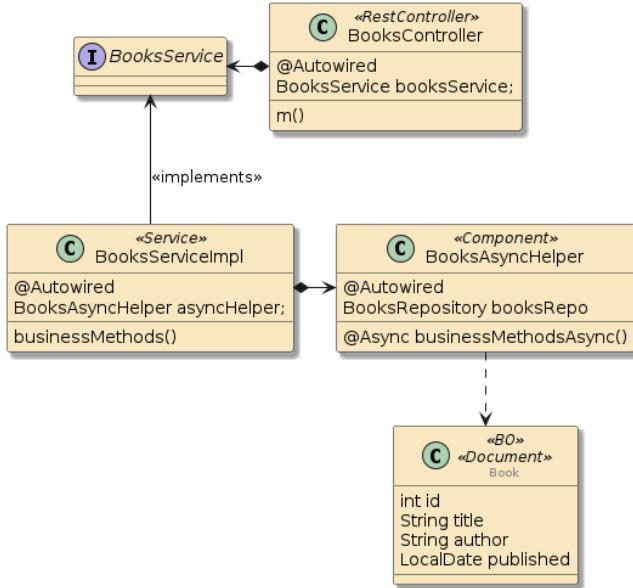


Figure 166. Single Service Expressed as Multiple Components

Each helper `@Component` is primarily designed around start and end of container augmentation. The remaining parts of the logical service are geared towards implementing the outward facing facade, and integrating the methods of the helper(s) to complete the intended role of the service. An example of this would be large loops of behavior.

```
for (...) { asyncHelper.asyncMethod(); }
```

To external users of `@Service`—it is still logically just one `@Service`.

Conceptual Services may be broken into Multiple Physical Components



Conceptual boundaries for a service usually map 1:1 with a single physical class. However, there are cases when the conceptual service needs to be implemented by multiple physical classes/`@Components`.

Chapter 369. BO/DTO Interface Options

With the core roles of BOs and DTOs understood, we next have a decision to make about where to use them within our application between the API and service classes.

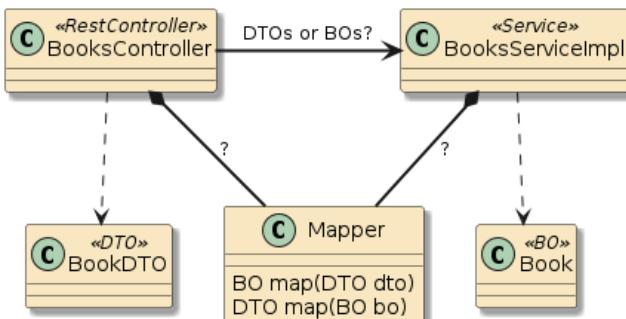


Figure 167. BO/DTO Interface Decisions

- @RestController external interface will always be based on DTOs.
- Service's internal implementation will always be based on BOs.
- Where do we make the transition?

369.1. API Maps DTO/BO

It is natural to think of the `@Service` as working with pure implementation (BO) classes. This leaves the mapping job to the `@RestController` and all clients of the `@Service`.

- Benefit: If we wire two `@Services` together, they could efficiently share the same BO instances between them with no translation.
- Drawback: `@Services` should be the boundary of a solution and encapsulate the implementation details. BOs leak implementation details.

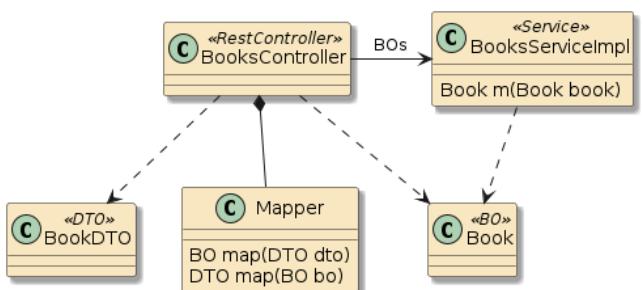


Figure 168. API Maps DTO to BO for Service Interface

369.2. @Service Maps DTO/BO

Alternatively, we can have the `@Service` fully encapsulate the implementation details and work with DTOs in its interface. This places the job of DTO/BO translation to the `@Service` and the `@RestController` and all `@Service` clients work with DTOs.

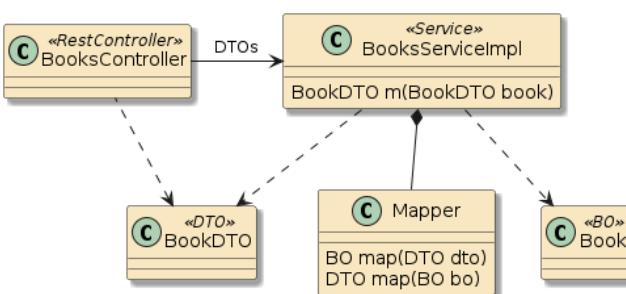
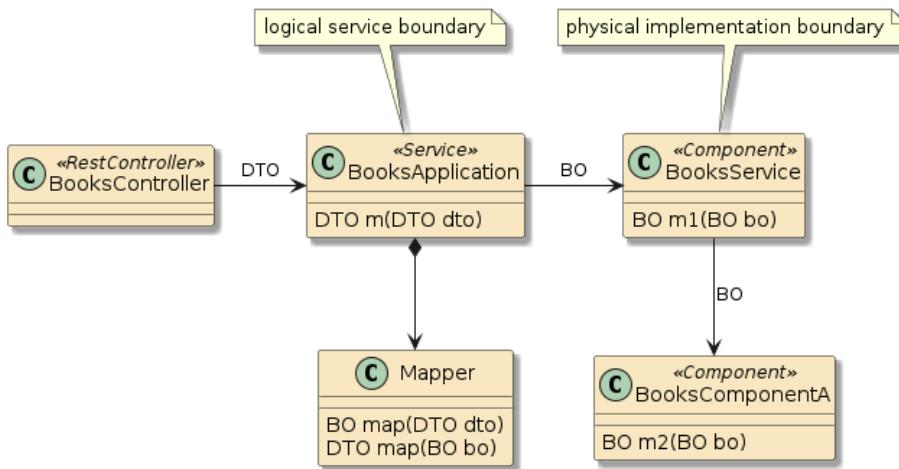


Figure 169. Service Maps DTO in Service Interface to BO

- Benefit: `@Service` fully encapsulates implementation and exchanges information using DTOs designed for interfaces.
- Drawback: BOs go through a translation when passing from `@Service` to `@Service` directly.

369.3. Layered Service Mapping Approach

The later DTO interface/mapping approach just introduced—maps closely to the [Domain Driven Design \(DDD\)](#) "Application Layer". However, one could also implement a layering of services.



- outer **@Service** classes represent the boundary to the application and interface using DTOs
- inner **@Component** classes represent implementation components and interface using native BOs

Layered Services Permit a Level of Trust between Inner Components

When using this approach, I like:



- all normalization and validation complete by the time DTOs are converted to BOs in the Application tier
- BOs exchanged between implementation components assume values are valid and normalized

Chapter 370. Implementation Details

With architectural decisions understood, lets take a look at some of the key details of the end-to-end application.

370.1. Book BO

We have already covered the `Book` BO `@Document` class in a lot of detail during the MongoTemplate lecture. The following lists most of the key business aspects and implementation details of the class.

Book BO Class with Spring Data MongoDB Database Mappings

```
package info.ejava.examples.db.mongo.books.bo;  
...  
@Document(collection = "books")  
@Builder  
@With  
@ToString  
@EqualsAndHashCode  
@Getter  
@Setter  
@AllArgsConstructorConstructor  
public class Book {  
    @Id @Setter(AccessLevel.NONE)  
    private String id;  
    @Setter  
    @Field(name="title")  
    private String title;  
    @Setter  
    private String author;  
    @Setter  
    private LocalDate published;  
}
```

370.2. BookDTO

The BookDTO class has been mapped to Jackson JSON and Jackson and JAXB XML. The details of Jackson and JAXB mapping were covered in the API Content lectures. Jackson JSON required no special annotations to map this class. Jackson and JAXB XML primarily needed some annotations related to namespaces and attribute mapping. JAXB also required annotations for mapping the LocalDate field.

The following lists the annotations required to marshal/unmarshal the BooksDTO class using Jackson and JAXB.

BookDTO Class with JSON and XML Mappings

```
package info.ejava.examples.db.repo.jpa.books.dto;
```

```

...
@JacksonXmlElement(localName = "book", namespace = "urn:ejava.db-repo.books")
@XmlRootElement(name = "book", namespace = "urn:ejava.db-repo.books")
@XmlAccessorType(XmlAccessType.FIELD)
@Data @Builder
@NoArgsConstructor @AllArgsConstructor
public class BookDTO {
    @JacksonXmlProperty(isAttribute = true)
    @XmlAttribute
    private int id;
    private String title;
    private String author;
    @XmlJavaTypeAdapter(LocalDateJaxbAdapter.class) ①
    private LocalDate published;
    ...
}

```

① JAXB requires an adapter for the newer LocalDate java class

370.2.1. LocalDateJaxbAdapter

Jackson is configured to marshal LocalDate out of the box using the ISO_LOCAL_DATE format for both JSON and XML.

ISO_LOCAL_DATE format

```

"published" : "2013-01-30" //Jackson JSON
<published xmlns="">2013-01-30</published> //Jackson XML

```

JAXB does not have a default format and requires the class be mapped to/from a string using an **XmlAdapter**.

LocalDateJaxbAdapter Class

```

@XmlJavaTypeAdapter(LocalDateJaxbAdapter.class) ①
private LocalDate published;

public static class LocalDateJaxbAdapter extends XmlAdapter<String, LocalDate> {②
    @Override
    public LocalDate unmarshal(String text) {
        return null!=text ? LocalDate.parse(text, DateTimeFormatter.ISO_LOCAL_DATE) :
    null;
    }
    @Override
    public String marshal(LocalDate timestamp) {
        return null!=timestamp ? DateTimeFormatter.ISO_LOCAL_DATE.format(timestamp) :
    null;
    }
}

```

- ① JAXB requires an adapter to translate from `LocalDate` to/from XML
- ② we can define an `XmlAdapter` to address `LocalDate` using java.time classes

370.3. Book JSON Rendering

The following snippet provides example JSON of a `Book` DTO payload.

Book JSON Rendering

```
{
  "id": "609b316de7366e0451a7bcb0",
  "title": "Tirra Lirra by the River",
  "author": "Mr. Arlen Swift",
  "published": "2020-07-26"
}
```

370.4. Book XML Rendering

The following snippets provide example XML of `Book` DTO payloads. They are technically equivalent from an XML Schema standpoint, but use some alternate syntax XML to achieve the same technical goals.

Book Jackson XML Rendering

```
<book xmlns="urn:ejava.db-repo.books" id="609b32b38065452555d612b8">
  <title>To a God Unknown</title>
  <author>Rudolf Harris</author>
  <published>2019-11-22</published>
</book>
```

Book JAXB XML Rendering

```
<ns2:book xmlns:ns2="urn:ejava.db-repo.books" id="609b32b38065452555d61222">
  <title>The Mermaids Singing</title>
  <author>Olen Rolfson IV</author>
  <published>2020-10-14</published>
</ns2:book>
```

370.5. Pageable/PageableDTO

I placed a high value on paging when working with unbounded collections when covering repository find methods. The value of paging comes especially into play when dealing with external users. That means we will need a way to represent Page, Pageable, and Sort in requests and responses as a part of DTO solution.

You will notice that I made a few decisions on how to implement this interface

1. I am assuming that both sides of the interface using the DTO classes are using Spring Data.
2. I am using the Page, Pageable, and Sort DTOs to directly self-map to/from Spring Data types. This makes the client and service code much simpler.

```
Pageable pageable = PageableDTO.of(pageNumber, pageSize, sortString).toPageable();  
①  
Page<BookDTO> result = ...  
BooksPageDTO resultDTO = new BooksPageDTO(result); ①
```

① using self-mapping between paging DTOs and Spring Data (`Pageable` and `Page`) types

3. I chose to use the Spring Data types in the `@Service` interface and performed the Spring Data to DTO mapping in the `@RestController`. I did this so that I did not eliminate any pre-existing library integration with Spring Data paging types.

```
Page<BookDTO> getBooks(Pageable pageable); ①
```

① using Spring Data (`Pageable` and `Page`) and business DTO (`BookDTO`) types in `@Service` interface

I will be going through the architecture and wiring in these lecture notes. The actual DTO code is surprisingly complex to render in the different formats and libraries. These topics were covered in detail in the API content lectures. I also chose to implement the `PageableDTO` and `sort` as immutable—which added some interesting mapping challenges worth inspecting.

370.5.1. PageableDTO Request

Requests require an expression for `Pageable`. The most straight forward way to accomplish this is through query parameters. The example snippet below shows `pageNumber`, `pageSize`, and `sort` expressed as simple string values as part of the URI. We have to write code to express and parse that data.

Example Pageable Query Parameters

```
①  
/api/books/example?pageNumber=0&pageSize=5&sort=published:DESC,id:ASC  
②
```

① `pageNumber` and `pageSize` are direct properties used by `PageRequest`

② `sort` contains a comma separated list of order compressed into a single string

Integer `pageNumber` and `pageSize` are straight forward to represent as numeric values in the query. `Sort` requires a minor amount of work. Spring Data `Sort` is an ordered list of property and direction. I have chosen to express property and direction using a ":" separated string and concatenate the ordering using a ",". This allows the query string to be expressed in the URI without special characters.

370.5.2. PageableDTO Client-side Request Mapping

Since I expect code using the `PageableDTO` to also be using Spring Data, I chose to use self-mapping between the `PageableDTO` and Spring Data `Pageable`.

The following snippet shows how to map `Pageable` to `PageableDTO` and the `PageableDTO` properties to URI query parameters.

Building URI with Pageable Request Parameters

```
PageRequest pageable = PageRequest.of(0, 5,
    Sort.by(Sort.Order.desc("published"), Sort.Order.asc("id")));
PageableDTO pageSpec = PageableDTO.of(pageable); ①
URI uri=UriComponentsBuilder
    .fromUri(serverConfig.getBaseUrl())
    .path(BooksController.BOOKS_PATH).path("/example")
    .queryParams(pageSpec.getQueryParams()) ②
    .build().toUri();
```

① using `PageableDTO` to self map from `Pageable`

② using `PageableDTO` to self map to URI query parameters

370.5.3. PageableDTO Server-side Request Mapping

The following snippet shows how the individual page request properties can be used to build a local instance of `PageableDTO` in the `@RestController`. Once the `PageableDTO` is built, we can use that to self map to a Spring Data `Pageable` to be used when calling the `@Service`.

```
public ResponseEntity<BooksPageDTO> findBooksByExample(
    @RequestParam(value="pageNumber",defaultValue="0",required=false) Integer pageNumber,
    @RequestParam(value="pageSize",required=false) Integer pageSize,
    @RequestParam(value="sort",required=false) String sortString,
    @RequestBody BookDTO probe) {

    Pageable pageable = PageableDTO.of(pageNumber, pageSize, sortString) ①
        .toPageable(); ②
```

① building `PageableDTO` from page request properties

② using `PageableDTO` to self map to Spring Data `Pageable`

370.5.4. Pageable Response

Responses require an expression for `Pageable` to indicate the pageable properties about the content returned. This must be expressed in the payload, so we need a JSON and XML expression for this. The snippets below show the JSON and XML DTO renderings of our `Pageable` properties.

Example JSON Pageable Response Document

```
"pageable" : {  
    "pageNumber" : 1,  
    "pageSize" : 25,  
    "sort" : "title:ASC,author:ASC"  
}
```

Example XML Pageable Response Document

```
<pageable xmlns="urn:ejava.common.dto" pageNumber="1" pageSize="25" sort=  
"title:ASC,author:ASC"/>
```

370.6. Page/PageDTO

Pageable is part of the overall `Page<T>`, with contents. Therefore, we also need a way to return a page of content to the caller.

370.6.1. PageDTO Rendering

JSON is very lenient and could have been implemented with a generic `PageDTO<T>` class.

```
{"content": [ ①  
    {"id":"609cffbc881de53b82657f17", ②  
     "title":"An Instant In The Wind",  
     "author":"Clifford Blick",  
     "published":"2003-04-09"}],  
    "totalElements":10, ①  
    "pageable":{"pageNumber":3,"pageSize":3,"sort":null}} ①
```

① `content`, `totalElements`, and `pageable` are part of reusable `PageDTO`

② book within `content` array is part of concrete Books domain

However, XML—with its use of unique namespaces, requires a sub-class to provide the type-specific values for content and overall page.

```
<booksPage xmlns="urn:ejava.db-repo.books" totalElements="10"> ①  
  <wstxns1:content xmlns:wstxns1="urn:ejava.common.dto">  
    <book id="609cffbc881de53b82657f17"> ②  
      <title xmlns="">An Instant In The Wind</title>  
      <author xmlns="">Clifford Blick</author>  
      <published xmlns="">2003-04-09</published>  
    </book>  
  </wstxns1:content>  
  <pageable xmlns="urn:ejava.common.dto" pageNumber="3" pageSize="3"/>  
</booksPage>
```

- ① `totalElements` mapped to XML as an (optional) attribute
- ② `booksPage` and `book` are in concrete domain `urn:ejava.db-repo.books` namespace

370.6.2. BooksPageDTO Subclass Mapping

The `BooksPageDTO` subclass provides the type-specific mapping for the content and overall page. The generic portions are handled by the base class.

BooksPageDTO Subclass Mapping

```

@JacksonXmlElementWrapper(localName="booksPage", namespace="urn:ejava.db-repo.books")
@XmlRootElement(name="booksPage", namespace="urn:ejava.db-repo.books")
@XmlType(name="BooksPage", namespace="urn:ejava.db-repo.books")
@XmlAccessorType(XmlAccessType.NONE)
@NoArgsConstructor
public class BooksPageDTO extends PageDTO<BookDTO> {
    @JsonProperty
    @JacksonXmlElementWrapper(localName="content", namespace="urn:ejava.common.dto")
    @JacksonXmlProperty(localName="book", namespace="urn:ejava.db-repo.books")
    @XmlElementWrapper(name="content", namespace="urn:ejava.common.dto")
    @XmlElement(name="book", namespace="urn:ejava.db-repo.books")
    public List<BookDTO> getContent() {
        return super.getContent();
    }
    public BooksPageDTO(List<BookDTO> content, Long totalElements,
                        PageableDTO pageableDTO) {
        super(content, totalElements, pageableDTO);
    }
    public BooksPageDTO(Page<BookDTO> page) {
        this(page.getContent(), page.getTotalElements(),
              PageableDTO.fromPageable(page.getPageable()));
    }
}

```

370.6.3. PageDTO Server-side Rendering Response Mapping

The `@RestController` can use the concrete DTO class (`BookPageDTO` in this case) to self-map from a Spring Data `Page<T>` to a DTO suitable for marshaling back to the API client.

PageDTO Server-side Response Mapping

```

Page<BookDTO> result=booksService.findBooksMatchingAll(probe, pageable);

BooksPageDTO resultDTO = new BooksPageDTO(result); ①
ResponseEntity<BooksPageDTO> response = ResponseEntity.ok(resultDTO);

```

- ① using `BooksPageDTO` to self-map Sing Data `Page<T>` to DTO

370.6.4. PageDTO Client-side Rendering Response Mapping

The `PageDTO<T>` class can be used to self-map to a Spring Data `Page<T>`. `Pageable`, if needed, can be obtained from the `Page<T>` or through the `pageDTO.getPageable()` DTO result.

PageDTO Client-side Response Mapping

```
//when
BooksPageDTO pageDTO = restTemplate.exchange(request, BooksPageDTO.class).getBody();
//then
Page<BookDTO> page = pageDTO.toPage(); ①
then(page.getSize()).isEqualTo(pageableRequest.getPageSize());
then(page.getNumber()).isEqualTo(pageableRequest.getPageNumber());
then(page.getSort()).isEqualTo(Sort.by(Sort.Direction.DESC, "published"));
Pageable pageable = pageDTO.getPageableDTO().toPageable(); ②
```

① using `PageDTO<T>` to self-map to a Spring Data `Page<T>`

② can use `page.getPageable()` or `pageDTO.getPageable().toPageable()` obtain `Pageable`

Chapter 371. BookMapper

The `BookMapper @Component` class is used to map between `BookDTO` and `Book` BO instances. It leverages Lombok builder methods — but is pretty much a simple/brute force mapping.

371.1. Example Map: BookDTO to Book BO

The following snippet is an example of mapping a `BookDTO` to a `Book` BO.

Map BookDTO to Book BO

```
@Component
public class BooksMapper {
    public Book map(BookDTO dto) {
        Book bo = null;
        if (dto!=null) {
            bo = Book.builder()
                .id(dto.getId())
                .author(dto.getAuthor())
                .title(dto.getTitle())
                .published(dto.getPublished())
                .build();
        }
        return bo;
    }
    ...
}
```

371.2. Example Map: Book BO to BookDTO

The following snippet is an example of mapping a `Book` BO to a `BookDTO`.

Map Book BO to BookDTO

```
...
public BookDTO map(Book bo) {
    BookDTO dto = null;
    if (bo!=null) {
        dto = BookDTO.builder()
            .id(bo.getId())
            .author(bo.getAuthor())
            .title(bo.getTitle())
            .published(bo.getPublished())
            .build();
    }
    return dto;
}
...
```

Chapter 372. Service Tier

The BooksService `@Service` encapsulates the implementation of our management of Books.

372.1. BooksService Interface

The `BooksService` interface defines a portion of pure CRUD methods and a series of finder methods. To be consistent with DDD encapsulation, the `@Service` interface is using DTO classes. Since the `@Service` is an injectable component, I chose to use straight Spring Data pageable types to possibly integrate with libraries that inherently work with Spring Data types.

BooksService Interface

```
public interface BooksService {  
    BookDTO createBook(BookDTO bookDTO); ①  
    BookDTO getBook(int id);  
    void updateBook(int id, BookDTO bookDTO);  
    void deleteBook(int id);  
    void deleteAllBooks();  
  
    Page<BookDTO> findPublishedAfter(LocalDate exclusive, Pageable pageable);②  
    Page<BookDTO> findBooksMatchingAll(BookDTO probe, Pageable pageable);  
}
```

① chose to use DTOs in `@Service` interface

② chose to use Spring Data types in pageable `@Service` finder methods

372.2. BooksServiceImpl Class

The `BooksServiceImpl` implementation class is implemented using the `BooksRepository` and `BooksMapper`.

BooksServiceImpl Implementation Attributes

```
@RequiredArgsConstructor ① ②  
@Service  
public class BooksServiceImpl implements BooksService {  
    private final BooksMapper mapper;  
    private final BooksRepository booksRepo;
```

① Creates a constructor for all final attributes

② Single constructors are automatically used for Autowiring

I will demonstrate two methods here — one that creates a book and one that finds books. There is no need for any type of formal transaction here because we are representing the boundary of consistency within a single document.

MongoDB 4.x Does Support Multi-document Transactions



[Multi-document transactions](#) are now supported within MongoDB (as of version 4.x) and [Spring Data MongoDB](#). When using declared transactions with Spring Data MongoDB, this looks identical to transactions implemented with Spring Data JPA. [The programmatic interface](#) is fairly intuitive as well. However, it is not considered a best, early practice. Therefore, I will defer that topic to a more advanced coverage of MongoDB interactions.

372.3. `createBook()`

The `createBook()` method

- accepts a `BookDTO`, creates a new book, and returns the created book as a `BookDTO`, with the generated ID.
- calls the mapper to map from/to a `BooksDTO` to/from a `Book BO`
- uses the `BooksRepository` to interact with the database

BooksServiceImpl.createBook()

```
public BookDTO createBook(BookDTO bookDTO) {
    Book bookBO = mapper.map(bookDTO); ①

    //insert instance
    booksRepo.save(bookBO); ②

    return mapper.map(bookBO); ③
}
```

① mapper converting DTO input argument to BO instance

② BO instance saved to database and updated with primary key

③ mapper converting BO entity to DTO instance for return from service

372.4. `findBooksMatchingAll()`

The `findBooksMatchingAll()` method

- accepts a `BookDTO` as a probe and `Pageable` to adjust the search and results
- calls the mapper to map from/to a `BooksDTO` to/from a `Book BO`
- uses the `BooksRepository` to interact with the database

BooksServiceImpl Finder Method

```
public Page<BookDTO> findBooksMatchingAll(BookDTO probeDTO, Pageable pageable) {
    Book probe = mapper.map(probeDTO); ①
    ExampleMatcher matcher = ExampleMatcher.matchingAll(); ②
    Page<Book> books = booksRepo.findAll(Example.of(probe, matcher), pageable); ③
```

```
    return mapper.map(books); ④  
}
```

- ① mapper converting DTO input argument to BO instance to create probe for match
- ② building matching rules to **AND** all supplied non-null properties
- ③ finder method invoked with matching and paging arguments to return page of BOs
- ④ mapper converting page of BOs to page of DTOs

Chapter 373. RestController API

The `@RestController` provides an HTTP Facade for our `@Service`.

`@RestController Class`

```
@RestController  
@Slf4j  
@RequiredArgsConstructor  
public class BooksController {  
    public static final String BOOKS_PATH="api/books";  
    public static final String BOOK_PATH= BOOKS_PATH + "/{id}";  
    public static final String RANDOM_BOOK_PATH= BOOKS_PATH + "/random";  
  
    private final BooksService booksService; ①
```

① `@Service` injected into class using constructor injection

I will demonstrate two of the operations available.

373.1. createBook()

The `createBook()` operation

- is called using `POST /api/books` method and URI
- passed a BookDTO, containing the fields to use marshaled in JSON or XML
- calls the `@Service` to handle the details of creating the Book
- returns the created book using a BookDTO

`createBook() API Operation`

```
@RequestMapping(path=BOOKS_PATH,  
    method=RequestMethod.POST,  
    consumes={MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE},  
    produces={MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE})  
public ResponseEntity<BookDTO> createBook(@RequestBody BookDTO bookDTO) {  
  
    BookDTO result = booksService.createBook(bookDTO); ①  
  
    URI uri = ServletUriComponentsBuilder.fromCurrentRequestUri()  
        .replacePath(BOOK_PATH)  
        .build(result.getId()); ②  
    ResponseEntity<BookDTO> response = ResponseEntity.created(uri).body(result);  
    return response; ③  
}
```

① DTO from HTTP Request supplied to and result DTO returned from `@Service` method

② URI of created instance calculated for `Location` response header

③ DTO marshalled back to caller with HTTP Response

373.2. `findBooksByExample()`

The `findBooksByExample()` operation

- is called using "POST /api/books/example" method and URI
- passed a BookDTO containing the properties to search for using JSON or XML
- calls the `@Service` to handle the details of finding the books after mapping the `Pageable` from query parameters
- converts the `Page<BookDTO>` into a `BooksPageDTO` to address marshaling concerns relative to XML.
- returns the page as a `BooksPageDTO`

findBooksByExample API Operation

```
@RequestMapping(path=BOOKS_PATH + "/example",
    method=RequestMethod.POST,
    consumes={MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE},
    produces={MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE})
public ResponseEntity<BooksPageDTO> findBooksByExample(
    @RequestParam(value="pageNumber",defaultValue="0",required=false) Integer
pageNumber,
    @RequestParam(value="pageSize",required=false) Integer pageSize,
    @RequestParam(value="sort",required=false) String sortString,
    @RequestBody BookDTO probe) {

    Pageable pageable=PageableDTO.of(pageNumber, pageSize, sortString).toPageable();①
    Page<BookDTO> result=booksService.findBooksMatchingAll(probe, pageable); ②

    BooksPageDTO resultDTO = new BooksPageDTO(result); ③
    ResponseEntity<BooksPageDTO> response = ResponseEntity.ok(resultDTO);
    return response;
}
```

① `PageableDTO` constructed from page request query parameters

② `@Service` accepts DTO arguments for call and returns DTO constructs mixed with Spring Data paging types

③ type-specific `BooksPageDTO` marshalled back to caller to support type-specific XML namespaces

373.3. WebClient Example

The following snippet shows an example of using a `WebClient` to request a page of finder results from the API. `WebClient` is part of the Spring WebFlux libraries—which implements reactive streams. The use of `WebClient` here is purely for example and not a requirement of anything created. However, using `WebClient` did force my hand to add JAXB to the DTO mappings since

Jackson XML is not yet supported by WebFlux. RestTemplate does support both Jackson and JAXB XML mapping - which would have made mapping simpler.

WebClient Client

```
@Autowired
private WebClient webClient;
...
UriComponentsBuilder findByExampleUriBuilder = UriComponentsBuilder
    .fromUri(serverConfig.getBaseUrl())
    .path(BooksController.BOOKS_PATH).path("/example");
...
//given
MediaType mediaType = ...
PageRequest pageable = PageRequest.of(0, 5, Sort.by(Sort.Order.desc("published")));
PageableDTO pageSpec = PageableDTO.of(pageable); ①
BookDTO allBooksProbe = BookDTO.builder().build(); ②
URI uri = findByExampleUriBuilder.queryParams(pageSpec.getQueryParams()) ③
    .build().toUri();
WebClient.RequestHeadersSpec<?> request = webClient.post()
    .uri(uri)
    .contentType(mediaType)
    .body(Mono.just(allBooksProbe), BookDTO.class)
    .accept(mediaType);
//when
ResponseEntity<BooksPageDTO> response = request
    .retrieve()
    .toEntity(BooksPageDTO.class).block();
//then
then(response.getStatusCode().is2xxSuccessful()).isTrue();
BooksPageDTO page = response.getBody();
```

① limiting query results to first page, ordered by "release", with a page size of 5

② create a "match everything" probe

③ pageable properties added as query parameters



WebClient/WebFlex does not yet support Jackson XML

WebClient and WebFlex does not yet support Jackson XML. This is what primarily forced the example to leverage JAXB for XML. WebClient/WebFlux automatically makes the decision/transition under the covers once an `@XmlRootElement` is provided.

Chapter 374. Summary

In this module, we learned:

- to integrate a Spring Data MongoDB Repository into an end-to-end application, accessed through an API
- implement a service tier that completes useful actions
- to make a clear distinction between DTOs and BOs
- to identify data type architectural decisions required for DTO and BO types
- to setup proper container feature boundaries using annotations and injection
- implement paging requests through the API
- implement page responses through the API

Heroku Database Deployments

copyright © 2024 jim stafford (jim.stafford@jhu.edu)

Chapter 375. Introduction

This lecture contains several "how to" aspects of building and deploying a Docker image to Heroku with Postgres or Mongo database dependencies.

375.1. Goals

You will learn:

- how to build a Docker image as part of the build process
- how to provision Postgres and Mongo internet-based resources for use with Internet deployments
- how to deploy an application to the Internet to use provisioned Internet resources

375.2. Objectives

At the conclusion of this lecture and related exercises, you will be able to:

1. provision a Postgres Internet-accessible database
2. provision a Mongo Internet-accessible database
3. map Heroku environment variables to Spring Boot properties using a shell script
4. build a Docker image as part of the build process

Chapter 376. Production Properties

We will want to use real database instances for remote deployment and we will get to that in a moment. For right now, lets take a look at some of the Spring Boot properties we need defined in order to properly make use of a live database.

376.1. Postgres Production Properties

We will need the following RDBMS properties individually enumerated for Postgres at runtime.

- `spring.data.datasource.url`
- `spring.data.datasource.username`
- `spring.data.datasource.password`

The remaining properties can be pre-set with a properties configuration embedded within the application.

Production Properties

```
##rdbms
#spring.datasource.url=... ①
#spring.datasource.username=...
#spring.datasource.password=...

spring.jpa.show-sql=false
spring.jpa.hibernate.ddl-auto=validate
spring.flyway.enabled=true
```

① datasource properties will be supplied at runtime

376.2. Mongo Production Properties

We will need the Mongo URL and luckily that and the user credentials can be expressed in a single URL construct.

- `spring.data.mongodb.uri`

There are no other mandatory properties to be set beyond the URL.

Production Properties

```
#mongo
#spring.data.mongodb.uri=mongodb://... ①
```

① `mongodb.uri` — with credentials — will be supplied at runtime

Chapter 377. Parsing Runtime Properties

The Postgres URL will be provided to us by Heroku using the `DATABASE_URL` property as shown below. They provide a means to separate the URL into variables, but that feature was not available for Docker deployments at the time I investigated. We can easily do that ourselves.

A logically equivalent Mongo URL will be made available from the Mongo resource provider. Luckily we can pass that single value in as the Mongo URL and be done.

Example Input Environment Variables

```
DATABASE_URL=postgres://postgres:secret@postgres:5432/postgres
MONGODB_URI=mongodb://admin:secret@mongo:27017/votes_db?authSource=admin
```

377.1. Environment Variable Script

Earlier—when PORT was the only thing we had to worry about—I showed a way to do that with the Dockerfile `CMD` option.

Review: Turning PORT Environment Variable into server.port Property

```
ENV PORT=8080
ENTRYPOINT ["java", "org.springframework.boot.loader.launch.JarLauncher"]
CMD ["--server.port=${PORT}"]
```

We could have expanded that same approach if we could get the `DATABASE_URL` broken down into URL and credentials. With that option not available, we can delegate to a script.

The following snippet shows the skeleton of the `run_env.sh` script we will put in place to address all types of environment variables we will see in our environments. The shell will launch whatever command was passed to it (`$(@)`) and append the `OPTIONS` that it was able to construct from environment variables. We will place this in the `src/docker` directory to be picked up by the Dockerfile.

The resulting script was based upon the much more complicated `example`.

run_env.sh Environment Variable Script

```
#!/bin/bash

OPTIONS=""

#ref: https://raw.githubusercontent.com/heroku/heroku-buildpack-jvm-
common/main/opt/jdbc.sh
if [[ -n "${DATABASE_URL:-}" ]]; then
    #
fi
```

```

if [[ -n "${MONGODB_URI:-}" ]]; then
    # ...
fi

if [[ -n "${PORT:-}" ]]; then
    # ...
fi

exec $@ ${OPTIONS}

```

377.2. Script Output

The following snippet shows an example `args` print of what is passed into the Spring Boot application from the `run_env.sh` script.

Resulting Command Line

```

args [--spring.datasource.url=jdbc:postgresql://postgres:5432/postgres,
--spring.datasource.username=postgres, --spring.datasource.password=secret,
--spring.data.mongodb.uri=mongodb://admin:secret@mongo:27017/votes_db?authSource=admin
]

```

Review: Remember that our environment will look like the following.

Input Environment Variables

```

DATABASE_URL=postgres://postgres:secret@postgres:5432/postgres
MONGODB_URI=mongodb://admin:secret@mongo:27017/votes_db?authSource=admin

```

Lets break down the details.

377.3. Heroku DataSource Property

The following script will breakout URL, username, and password and turn them into Spring Boot properties on the command line.

DataSource Properties

```

if [[ -n "${DATABASE_URL:-}" ]]; then
    pattern="^postgres://(.+):(.+)@(.+)$" ①
    if [[ "${DATABASE_URL}" =~ $pattern ]]; then ②
        JDBC_DATABASE_USERNAME="${BASH_REMATCH[1]}"
        JDBC_DATABASE_PASSWORD="${BASH_REMATCH[2]}"
        JDBC_DATABASE_URL="jdbc:postgresql://${BASH_REMATCH[3]}"

        OPTIONS="${OPTIONS} --spring.datasource.url=${JDBC_DATABASE_URL} "
        OPTIONS="${OPTIONS} --spring.datasource.username=${JDBC_DATABASE_USERNAME}"
        OPTIONS="${OPTIONS} --spring.datasource.password=${JDBC_DATABASE_PASSWORD}"

```

```

else
    OPTIONS="${OPTIONS} --no.match=${DATABASE_URL}" ③
fi
fi

```

- ① regular expression defining three (3) extraction variables
- ② if the regular expression finds a match, we will pull that apart and assemble the properties
- ③ if no match is found, `--no.match` is populated with the DATABASE_URL to be printed for debug reasons

377.4. Testing DATABASE_URL

You can test the script so far by invoking the with the environment variable set.

Testing Postgres URL Parsing

```
(export DATABASE_URL=postgres://postgres:secret@postgres:5432/postgres && bash
./src/docker/run_env.sh echo)
```

Expected Postgres Output

```
--spring.datasource.url=jdbc:postgresql://postgres:5432/postgres
--spring.datasource.username=postgres --spring.datasource.password=secret
```

Of course, that same test could be done with a Docker image.

Testing Postgres URL Parsing within Docker

```
docker run --rm \
-e DATABASE_URL=postgres://postgres:secret@postgres:5432/postgres \①
-v `pwd`/src/docker/run_env.sh:/tmp/run_env.sh \②
openjdk:17.0.2 \
/tmp/run_env.sh echo ③
```

- ① setting the environment variable
- ② mounting the file in the `/tmp` directory
- ③ running script and passing in `echo` as executable to call

377.5. MongoDB Properties

The Mongo URL we get from Atlas can be passed in as a single property. If Postgres was this straight forward, we could have stuck with the `CMD` option.

MongoDB Property

```
if [[ -n "${MONGODB_URI:-}" ]]; then
    OPTIONS="${OPTIONS} --spring.data.mongodb.uri=${MONGODB_URI}"
```

```
fi
```

Demonstrating Mongo URL Handling

```
(export MONGODB_URI=mongodb://admin:secret@mongo:27017/votes_db?authSource=admin &&
bash ./src/docker/run_env.sh echo)
```

Expected Mongo Output

```
--spring.data.mongodb.uri=mongodb://admin:secret@mongo:27017/votes_db?authSource=admin
```

377.6. PORT Property

We need to continue supporting the **PORT** environment variable and will add a block for that.

Server Port Property

```
if [[ -n "${PORT:-}" ]]; then
  OPTIONS="${OPTIONS} --server.port=${PORT}"
fi
```

Testing All Together

```
(export DATABASE_URL=postgres://postgres:secret@postgres:5432/postgres && export
MONGODB_URI=mongodb://admin:secret@mongo:27017/votes_db?authSource=admin && export
PORT=7777 && bash ./src/docker/run_env.sh echo)
```

Expected Aggregate Output

```
--spring.datasource.url=jdbc:postgresql://postgres:5432/postgres
--spring.datasource.username=postgres --spring.datasource.password=secret
--spring.data.mongodb.uri=mongodb://admin:secret@mongo:27017/votes_db?authSource=admin
--server.port=7777
```

Chapter 378. Docker Image

With the embedded properties set, we are now ready to build a Docker image. We will use a Maven plugin to build the image using Docker since the memory requirement for the default Spring Boot Docker image exceeds the Heroku Memory limit for free deployments.

378.1. Dockerfile

The following shows the Dockerfile being used. It is 99% of what can be found in the Spring Boot Maven Plugin Documentation except for:

- a tweak on the `ARG JAR_FILE` command to add our `bootexec` classifier. Note that our local Maven `pom.xml` `JAR_FILE` declaration will take care of this as well.
- `src/docker/run_env.sh` script added to search for environment variables and break them down into Spring Boot properties

Example Dockerfile

```
FROM openjdk:17.0.2 as builder
WORKDIR application
ARG JAR_FILE=target/*-bootexec.jar ①
COPY ${JAR_FILE} application.jar
RUN java -Djarmode=layer-tools -jar application.jar extract

FROM openjdk:17.0.2
WORKDIR application
COPY --from=builder application/dependencies/ ./
COPY --from=builder application/spring-boot-loader/ ./
COPY --from=builder application/snapshot-dependencies/ ./
COPY --from=builder application/application/ ./
COPY src/docker/run_env.sh ./ ②
RUN chmod +x ./run_env.sh
ENTRYPOINT ["../run_env.sh",
"java","org.springframework.boot.loader.launch.JarLauncher"]
```

① Spring Boot executable JAR has `bootexec` Maven `classifier` suffix added

② added a filter script to break certain environment variables into separate properties

378.2. Spotify Docker Build Maven Plugin

At this point with a Dockerfile in hand, we have the option of building the image with straight `docker build` or `docker-compose build`. We can also use the Spotify Docker Maven Plugin to automate the build of the Docker image as part of the module build. The plugin is forming an explicit path to the JAR file and using the `JAR_FILE` variable to pass that into the `Dockerfile`. Note that by supplying the `JAR_FILE` reference here, we can build images without worrying about the wildcard glob in the Dockerfile locating too many matches.

Spotify Docker Build Maven Plugin

```
<plugin>
  <groupId>com.spotify</groupId>
  <artifactId>dockerfile-maven-plugin</artifactId>
  <configuration>
    <repository>${project.artifactId}</repository>
    <tag>${project.version}</tag>
    <buildArgs>
      <JAR_FILE>target/${project.build.finalName}-${spring-boot.classifier}.jar</JAR_FILE>
      ①
        </buildArgs>
    </configuration>
    <executions>
      <execution>
        <goals>
          <goal>build</goal>
        </goals>
      </execution>
    </executions>
  </plugin>
```

① JAR_FILE is passed in as a build argument to Docker

Spotify Docker Build Maven Plugin Completing Build

```
[INFO] Successfully built dfe2383f7f68
[INFO] Successfully tagged xxx:6.1.0-SNAPSHOT
[INFO]
[INFO] Detected build of image with id dfe2383f7f68
...
[INFO] Successfully built dockercompose-votes-svc:6.1.0-SNAPSHOT
[INFO] -----
[INFO] BUILD SUCCESS
```

Chapter 379. Heroku Deployment

The following are the basic steps taken to deploy the Docker image to Heroku.

379.1. Provision MongoDB

MongoDB offers a Mongo database service on the Internet called [Atlas](#). They offer free accounts and the ability to setup and operate database instances at no cost.

- Create account using email address
- Create a new project
- Create a new (free) cluster within that project
- Create username/password for DB access
- Setup Internet IP whitelist (can be wildcard/all) of where to accept connects from. I normally set that to everywhere — at least until I locate the Heroku IP address.
- Obtain a URL to connect to. It will look something like the following:

```
mongodb+srv://(username):(password)@(host)/(dbname)?retryWrites=true&w=majority
```

379.2. Provision Application

Refer back to the Heroku lecture for details, but essentially

- create a new application
- set the MONGODB_URI environment variable for that application
- set the SPRING_PROFILES_ACTIVE environment variable to `production`

```
$ heroku create [app-name]
$ heroku config:set MONGODB_URI=mongodb+srv://(username):(password)@(host)/votes_db...
--app (app-name)
$ heroku config:set SPRING_PROFILES_ACTIVE=production
```

379.3. Provision Postgres

We can provision Postgres directly on Heroku itself.

Example Postgres Provision

```
$ heroku addons:create heroku-postgresql:hobby-dev
Creating heroku-postgresql:hobby-dev on 0 xxx... free
Database has been created and is available
! This database is empty. If upgrading, you can transfer
! data from another database with pg:copy
```

```
Created postgresql-shallow-xxxxx as DATABASE_URL
Use heroku addons:docs heroku-postgresql to view documentation
```

After the provision, we can see that a compound DATABASE_URI was provided

```
$ heroku config --app app-name
== app-name Config Vars
DATABASE_URL: postgres://(username):(password)@(host):(port)/(database)
MONGODB_URI: mongodb+srv://(username):(password)@(host)/votes_db?...
SPRING_PROFILES_ACTIVE: production
```

379.4. Deploy Application

Tag Docker Image

```
$ docker tag (artifactId):(tag) registry.heroku.com/(app-name)/web
```

Push Docker Image Using Tag

```
$ heroku container:login
Login Succeeded
$ docker push registry.heroku.com/(app-name)/web
The push refers to repository [registry.heroku.com/(app-name)/web]
6f38c0466979: Pushed
69a39355b3ac: Pushed
ea12a8cf9f94: Pushed
d2451ff7adf4: Layer already exists
...
7ef368776582: Layer already exists
latest: digest:
sha256:21197b193a6657dd5e6f10d6751f08faa416a292a17693ac776b211520d84d19 size: 3035
```

379.5. Release the Application

Invoke the Heroku release command to make the changes visible to the Internet.

Make Application Available

```
$ heroku container:release web --app (app-name)
Releasing images web to (app-name)... done
```

Tail the Heroku log to verify the application starts and the production profile is active.

Tail Heroku Log

```
$ heroku logs --app (app-name) --tail
```

```
/\\ / ___ ' - -- - - ( ) - -- - - \ \\ \\  
( ( )\___ | ' _ | ' _ | ' _ \V _ ' | \ \\ \\  
\\V_ ___) | _)| | | | | | ( _| | ) ) ) )  
' | ___| . _| _| _| _| _\_, | / / / /  
=====|_|=====|_|/_=/_/_/_/  
:: Spring Boot ::      (3.3.2)
```

The following profiles are active: production ①

① make sure the application is running the correct profile

Chapter 380. Summary

In this module, we learned:

- how to provision internet-based MongoDB and Postgres resources
- how to deploy an application to the Internet to use provisioned Postgres and Mongo database resources
- how to build a Docker image as part of the build process

AutoRentals Assignment 5: DB

copyright © 2024 jim stafford (jim.stafford@jhu.edu)

This assignment is broken up into three mandatory sections.

The first two mandatory sections functionally work with Spring Data Repositories outside the scope of the AutoRentals workflow. You will create a "service" class that is a peer to your AutoRentals Service implementation—but this new class is there solely as a technology demonstration and wrapper for the provided JUnit tests. You will work with both JPA and Mongo Repositories as a part of these first two sections.

In the third mandatory section—you will select one of the two technologies, update the end-to-end thread with a Spring Data Repository, and add in some Pageable and Page aspects for unbounded collection query/results.

Chapter 381. Assignment 5a: Spring Data JPA

381.1. Database Schema

381.1.1. Purpose

In this portion of the assignment, you will demonstrate your knowledge of preparing a relational database for use with an application. You will:

1. define a database schema that maps a single class to a single table
2. implement a primary key for each row of a table
3. define constraints for rows in a table
4. define an index for a table
5. define a DataSource to interface with the RDBMS
6. automate database schema migration with the Flyway tool

381.1.2. Overview

In this portion of the assignment you will be defining, instantiating, and performing minor population of a database schema for AutoRental. We will use a single, flat database design.

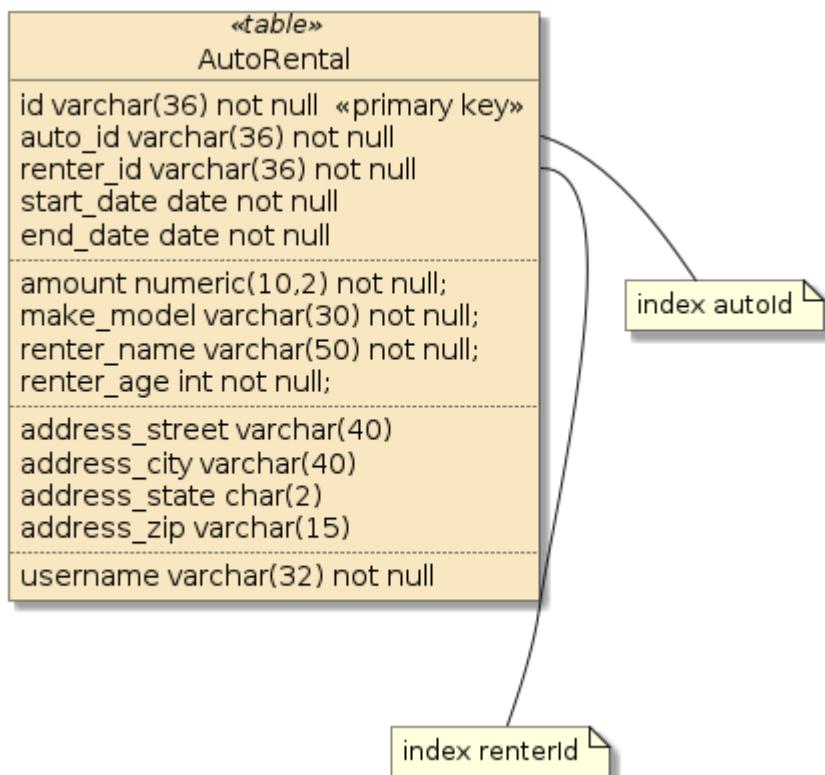


Figure 170. AutoRentals Schema

The assignment will use a 36-character UUID for the ID. No sequence will be necessary.



Use varchar(36) to allow consistency with Mongo portion of assignment

Use the varchar-based primary key to make the JPA and Mongo portions of the assignment as similar as possible. We will use a UUID for the JPA portion, but any unique String fitting into 36 characters will work.

Postgres access with Docker/Docker Compose

If you have Docker/Docker Compose, you can instantiate a Postgres instance using the scripts in the ejava-springboot root directory.

```
$ docker-compose up -d postgres
Creating network "ejava_default" with the default driver
Creating ejava_postgres_1 ... done
```



You can also get client access to using the following command.

```
$ docker-compose exec postgres psql -U postgres
psql (12.3)
Type "help" for help.

postgres=#
```

You can switch between in-memory H2 (default) and Postgres once you have your property files setup either by manual change of the source code or using runtime properties with the `TestProfileResolver` class provided in the starter module. You may also leverage Testcontainers to stick with the same database at all times.



```
@SpringBootTest(...)
@ActiveProfiles(profiles={"assignment-tests", "test"}, resolver =
TestProfileResolver.class)
//@ActiveProfiles(profiles={"assignment-tests", "test", "postgres"})
public class Jpa5a_SchemaTest {
```

381.1.3. Requirements

1. Create a set of SQL migrations below `src/main/resources/db/migration` that will define the database schema



Refer to the [JPA songs example](#) for a schema example. However, that example assumes that all schema is vendor-neutral and does not use vendor-specific sibling files.

- a. create SQL migration file(s) to define the base AutoRental schema. This can be hand-generated or metadata-generated once the `@Entity` class is later defined
 - i. account for when the table(s)/sequence(s) already exist by defining a DROP before creating



```
drop table IF EXISTS (table name);
```

- ii. define a AutoRental table with the necessary columns to store a flattened AutoRental object. We are going to store every property of an AutoRental and its 1:1 relationship(s) in the same table.
 - A. use the `id` field as a primary key. Make this a char-based column type of at least 36 characters (`varchar(36)`) to host a UUID string
 - B. define column constraints for size and not-null
- b. Create a separate SQL migration file to add indexes
 - i. define a non-unique index on `auto_id`
 - ii. define a non-unique index on `renter_id`
- c. Create a SQL migration file to add one sample row in the AutoRental table



`CURRENT_DATE` (`CURRENT_DATE+7`) can be used to generate a value for dates

You can manually test schema files by launching the Postgres client and reading the SQL file in from stdin



Execute SQL File Against Postgres using Docker Compose

```
docker-compose exec -T postgres psql -U postgres < (path to  
file)
```

- d. Place vendor-neutral SQL in a `common` and vendor-specific SQL in a `{vendor}` directory as defined in your flyway properties. The example below shows a possible layout.

```
src/main/resources/  
`-- db  
    '-- migration  
        |-- common  
        |-- h2  
        '-- postgres
```



I am not anticipating any vendor-specific schema population, but it is a good practice if you use multiple database vendors between development and production.



If you have Postgres running, you can test your schema with the following:

```
$ docker-compose -f .../docker-compose.yml exec -T postgres psql -U  
postgres <  
.src/main/resources/db/migration/common/V1.0.0_0__autorentals_schem
```

```

a.sql
NOTICE: table "rentals_autorental" does not exist, skipping
DROP TABLE
CREATE TABLE
COMMENT
COMMENT
...

$ docker-compose -f .../docker-compose.yml exec -T postgres psql -U
postgres <
./src/main/resources/db/migration/common/V1.0.0_1__autorentals_index
es.sql
CREATE INDEX
CREATE INDEX

$ docker-compose -f .../docker-compose.yml exec -T postgres psql -U
postgres <
./src/main/resources/db/migration/postgres/V1.0.0_2__autorentals_pop
ulate.sql
INSERT 0 1
UPDATE 1

$ docker-compose -f .../docker-compose.yml exec postgres psql -U
postgres -c '\d'
      List of relations
 Schema |           Name           | Type  | Owner
-----+-----+-----+-----+
 public | (table name) | table | postgres
(1 row)

$ docker-compose -f .../docker-compose.yml exec postgres psql -U
postgres -c '\d (table name)'
      Column     |          Type          | Collation | Nullable | Default
-----+-----+-----+-----+
      id        | character varying(36) |          | not null |
 auto_id    | character varying(36) |          | not null |
 renter_id  | character varying(36) |          | not null |
 ...
Indexes:
...
$ docker-compose -f .../docker-compose.yml exec postgres psql -U
postgres -c '\d+ (table name)'

```

- Configure database properties so that you are able to work with both in-memory and external database. In-memory will be good for automated testing. Docker Compose-launched Postgres will be good for interactive access to the database while developing. Testcontainers-based

Postgres can also be used but can be harder to work with for interactive development.

- a. make the default database in-memory

You can set the default database to `h2` and activate the console by setting the following properties.



application.properties

```
#default test database
spring.datasource.url=jdbc:h2:mem:autorentals
spring.h2.console.enabled=true
```

- b. provide a "postgres" Spring profile option to use Postgres DB instead of in-memory

You can switch to an alternate database by overriding the URL in a Spring profile. Add a `postgres` Spring profile in `src/main` tree to optionally connect to an external Postgres server versus the in-memory H2 server. Include any necessary credentials. The following example assumes you will be connected to the Postgres DB launched by the class docker-compose.



application-postgres.properties

```
spring.datasource.url=jdbc:postgresql://localhost:5432/postgres
spring.datasource.username=postgres
spring.datasource.password=secret
```



This is only a class assignment. Do not store credentials in files checked into CM or packaged within your Spring Boot executable JAR in a real environment. Make them available via a file location at runtime when outside of a classroom.

- c. define the location for your schema migrations for flyway to automate.



```
spring.flyway.locations=classpath:db/migration/common,classpath:
db/migration/{vendor}
```

- 3. Configure the application to establish a connection to the database and establish a DataSource

- a. declare a dependency on `spring-boot-starter-data-jpa`
- b. declare a dependency on the `h2` database driver for default testing and demonstration
- c. declare a dependency on the `postgresql` database driver for optional production-ready testing
- d. declare the database driver dependencies as `scope=runtime`



See [jpa-song-example pom.xml](#) for more details on declaring these

dependencies.

4. Configure Flyway so that it automatically populates the database schema
 - a. declare a dependency on the `flyway-core` schema migration library
 - b. declare the Flyway dependency as `scope=runtime`



See [jpa-song-example pom.xml](#) for more details on declaring this plugin

5. Enable (and pass) the provided `MyJpa5a_SchemaTest` that extends `Jpa5a_SchemaTest`. This test will verify connectivity to the database and the presence of the `AutoRental` table.
 - a. supply necessary `@SpringBootTest` test configurations unique to your environment
 - b. supply an implementation of the `DbTestHelper` to be injected into all tests

If the `MyJpa5a_SchemaTest` is launched without a web environment, Spring's AutoConfiguration will not provide web-related components for injection (e.g., `RestTemplate`). The web environment is unnecessary for this and most of the database portions of the assignment. Design your test helper(s) and test helper `@Bean` factories so that they can detect the absence of unnecessary web-related components and ignore them.



Example Detection of non-Web Environment

```
@Bean @Lazy  
@ConditionalOnBean({RestTemplate.class, ServerConfig.class})  
public DbTestHelper fullTestHelper(/*web-related resources*/) {  
    @Bean  
    @ConditionalOnMissingBean({RestTemplate.class, ServerConfig.class})  
    public DbTestHelper dbOnlyTestHelper() {
```

6. Package the JUnit test case such that it executes with Maven as a surefire test

381.1.4. Grading

Your solution will be evaluated on:

1. define a database schema that maps a single class to a single table
 - a. whether you have expressed your database schema in one or more files
2. implement a primary key for each row of a table
 - a. whether you have identified the primary key for the table
3. define constraints for rows in a table
 - a. whether you have defined size and nullable constraints for columns
4. define an index for a table
 - a. whether you have defined an index for any database columns

5. automate database schema migration with the Flyway tool
 - a. whether you have successfully populated the database schema from a set of files
6. define a DataSource to interface with the RDBMS
 - a. whether a DataSource was successfully injected into the JUnit class

381.1.5. Additional Details

1. This and the following RDBMS/JPA and MongoDB tests are all client-side DB interaction tests. Calls from JUnit are directed at the service class. The provided starter example supplies an alternate `@SpringBootConfiguration` test configuration to bypass the extra dependencies defined by the full `@SpringBootApplication` server class—which can cause conflicts. The `@SpringBootConfiguration` class is latched by the "assignment-tests" profile to keep it from being accidentally used by the later API tests.

```
@SpringBootConfiguration
@EnableAutoConfiguration
@Profile("assignment-tests") ①
public class DbAssignmentTestConfiguration {

    @SpringBootTest(classes={DbAssignmentTestConfiguration.class,
        JpaAssignmentDBConfiguration.class,
        DbClientTestConfiguration.class})
    @ActiveProfiles(profiles={"assignment-tests", "test"}, resolver =
    TestProfileResolver.class)②
    //@ActiveProfiles(profiles={"assignment-tests", "test", "postgres"})
    @Slf4j
    public class MyJpa5a_SchemaTest extends Jpa5a_SchemaTest {
```

① profile prevents `@SpringBootConfiguration` from being used as a `@Configuration` for other tests
 ② `assignment-tests` profile is activated for these service/DB-level tests only

2. The following `starter` configuration files are used by the tests in this section:
 - a. `DbAssignmentTestConfiguration` - discussed above. Provides a `@SpringBootConfiguration` class that removes the `@SpringBootApplication` dependencies from view.
 - b. `DbClientTestConfiguration` - this defines the `@Bean` factories for the `DbTestHelper` and any supporting components.
 - c. `JpaAssignmentDBConfiguration` - this defines server-side beans used in this DB-centric portion of the assignment. It provides `@Bean` factories that will get replaced when running the application and performing the end-to-end tests.

381.2. Entity/BO Class

381.2.1. Purpose

In this portion of the assignment, you will demonstrate your knowledge of defining a JPA `@Entity` class and performing basic CRUD actions. You will:

1. define a PersistenceContext containing an `@Entity` class
2. inject an EntityManager to perform actions on a Persistence Unit and database
3. map a simple `@Entity` class to the database using JPA mapping annotations
4. perform basic database CRUD operations on an `@Entity`
5. define transaction scopes
6. implement a mapping tier between BO and DTO objects

381.2.2. Overview

In this portion of the assignment you will be creating an `@Entity/Business Object` for a AutoRental, mapping that to a table, and performing CRUD actions with an EntityManager.

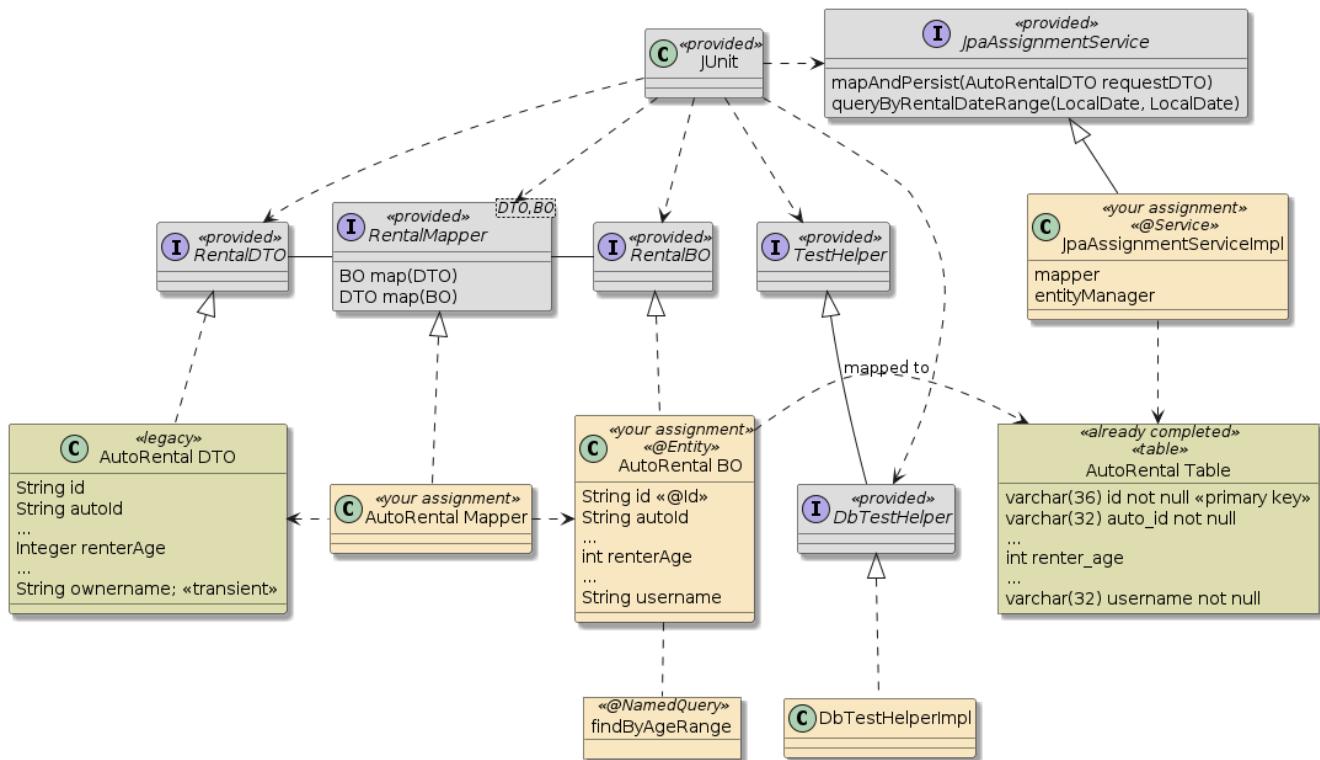


Figure 171. AutoRental Entity

Your work will be focused in the following areas:

- creating a business object (BO)`@Entity` class to map to the database schema you have already completed
- creating a mapper class that will map properties to/from the DTO and BO instances
- creating a test helper class, implementing `DbTestHelper` that will assist the provided JUnit tests to interact and inspect your persistence implementation.
- implementing a `JpaAssignmentService` component that will perform specific interactions with

the database

The interfaces for the `DbTestHelper` and `JpaAssignmentService` are located in the support module containing the tests. The `DbTestHelper` interface extends the `ApiTestHelper` interface you have previously implemented and will simply extend with the additional functionality.

The BO and mapper classes will be used throughout this overall assignment, including the end-to-end. The testHelper class will be used for all provided JUnit tests. The `JpaAssignmentService` will only be used during the JPA-specific sections of this assignment. It is a sibling to your `AutoRentalsService` component(s) for the purpose of doing one-off database assignment tasks. It will not be used in the Mongo portions of the assignment or the end-to-end portion.

381.2.3. Requirements

1. Create a Business Object (BO)/`@Entity` class that represents the AutoRental and will be mapped to the database. A `RentalBO` "marker" interface has been provided for your BO class to implement. It has no properties. All interactions with this object by the JUnit test will be through calls to the testHelper and mapper classes. You must complete the following details:
 - a. identify the class as a JPA `@Entity`
 - b. map the class to the DB table you created in the previous section using the `@Table` annotation.
 - c. identify a String primary key field with JPA `@Id`
 - d. supply a default constructor
 - e. map the attributes of the class to the columns you created in the previous section using the `@Column` annotation.
 - f. supply other constructs as desired to help use and interact with this business object



The BO class will map to a single, flat database row. Keep that in mind when accounting for the address. The properties/structure of the BO class do not have to be 1:1 with the properties/structure of the DTO class.

- g. supply a lifecycle event handler that will assign the string representation of a UUID to the `id` field if null when persisted

`@Entity @PrePersist Lifecycle Callback to assign Primary Key`

```
import jakarta.persistence.PrePersist;  
  
    @PrePersist  
    void prePersist() {  
        if (id==null) {  
            id= UUID.randomUUID().toString();  
        }  
    }
```



If your Entity class is not within the default scan path, you can manually register the package path using the `@EntityScan.basePackageClasses`

annotation property. This should be done within a `@Configuration` class in the `src/main` portion of your code. The JUnit test will make the condition and successful correction obvious.

2. Create a mapper class that will map to/from AutoRental BO and DTO. A templated `RentalsMapper` interface has been provided for this. You must complete the details.
 - a. map from BO to DTO
 - b. map from DTO to BO



Remember — the structure of the BO and DTO classes do not have to match. Encapsulate any mapping details between the two within this mapper class implementation.

The following code snippet shows an example implementation of the templated mapper interface.



```
public class AutoRentalMapper implements RentalMapper<AutoRentalDTO, AutoRentalBO> {  
    public AutoRentalBO map(AutoRentalDTO dto) { ... }  
    public AutoRentalDTO map(AutoRentalBO bo) { ... }
```

- c. expose this class as a component so that it can be injected by the tests.

Use the `JpaAssignmentDBConfiguration` class used by the JUnit test to expose this through a `@Bean` factory.



```
@SpringBootTest(classes={DbAssignmentTestConfiguration.class,  
    JpaAssignmentDBConfiguration.class,  
    DbClientTestConfiguration.class})  
@ActiveProfiles(profiles={"assignment-tests","test"}, resolver =  
TestProfileResolver.class)  
...  
public class MyJpa5b_EntityTest extends Jpa5b_EntityTest {
```

3. Implement a class that implements the `JpaAssignmentService<DTO, BO>` interface.
 - a. Instantiate this as a component in the "assignment-tests" profile.
4. Implement the `mapAndPersist` method in your `JpaAssignmentService`. It must perform the following:
 - a. accept a AutoRental DTO
 - b. map the DTO to a AutoRental BO (using your mapper)
 - c. persist the BO
 - d. map the persisted BO to a DTO (will have a primary key assigned)

- e. return the resulting DTO

The BO must be persisted. The returned DTO must match the input DTO and express the primary key assigned by the database.



Be sure to address `@Transactional` details when modifying the database.

5. Implement the `queryByRentalDateRange` method in the `JpaAssignmentService` using a `@NamedQuery`. It must perform the following:

- a. query the database using a JPA `@NamedQuery` with JPA query syntax to locate `AutoRental` BO objects within a `startDate/endDate` range, inclusive

You may use the following query to start with and add ordering to complete



```
select r from AutoRentalBO r where startDate <= :endDate and  
endDate <= :startDate
```

- i. `:startDate` and `:endDate` are variable `LocalDate` values passed in at runtime with `setParameter()`
- ii. order the results by `id` ascending
- iii. name the query "`<EntityName>.findByDatesBetween`"



`EntityName` defaults to the Java SimpleName for the class. Make sure all uses of `EntityName` (i.e., JPQL queries and JPA `@NamedQuery` name prefixes) match.

- b. map the BO list returned from the query to a list of DTOs (using your mapper)
- c. return the list of DTOs
6. Enable (and pass) the provided `MyJpa5b_EntityTest` that extends `Jpa5b_EntityTest`. This test will perform checks of the above functionality using:
 - a. `DbTestHelper`
 - b. mapper
 - c. your DTO and BO classes
 - d. a functional JPA environment
7. Package the JUnit test case such that it executes with Maven as a surefire test

381.2.4. Grading

Your solution will be evaluated on:

1. inject an `EntityManager` to perform actions on a Persistence Unit and database
 - a. whether an `EntityManager`/`EntityManager` was successfully injected into the JUnit test
 - b. whether an `EntityManager` was successfully injected into your `JpaAssignmentService`

- implementation
2. map a simple `@Entity` class to the database using JPA mapping annotations
 - a. whether a new AutoRental BO class was created for mapping to the database
 - b. whether the class was successfully mapped to the database table and columns
 3. implement a mapping tier between BO and DTO objects
 - a. whether the mapper was able to successfully map all fields between BO to DTO
 - b. whether the mapper was able to successfully map all fields between DTO to BO
 4. perform basic database CRUD operations on an `@Entity`
 - a. whether the AutoRental BO was successfully persisted to the database
 - b. whether a named JPA-QL query was used to locate the entity in the database
 5. define transaction scopes
 - a. whether the test method was declared to use a single transaction for all steps of the test method

381.2.5. Additional Details

381.3. JPA Repository

381.3.1. Purpose

In this portion of the assignment, you will demonstrate your knowledge of defining a JPA Repository. You will:

1. declare a `JpaRepository` for an existing JPA `@Entity`
2. perform simple CRUD methods using provided repository methods
3. add paging and sorting to query methods
4. implement queries based on predicates derived from repository interface methods
5. implement queries based on POJO examples and configured matchers
6. implement queries based on `@NamedQuery` or `@Query` specification

381.3.2. Overview

In this portion of the assignment, you will define a JPA Repository to perform basic CRUD and query actions.

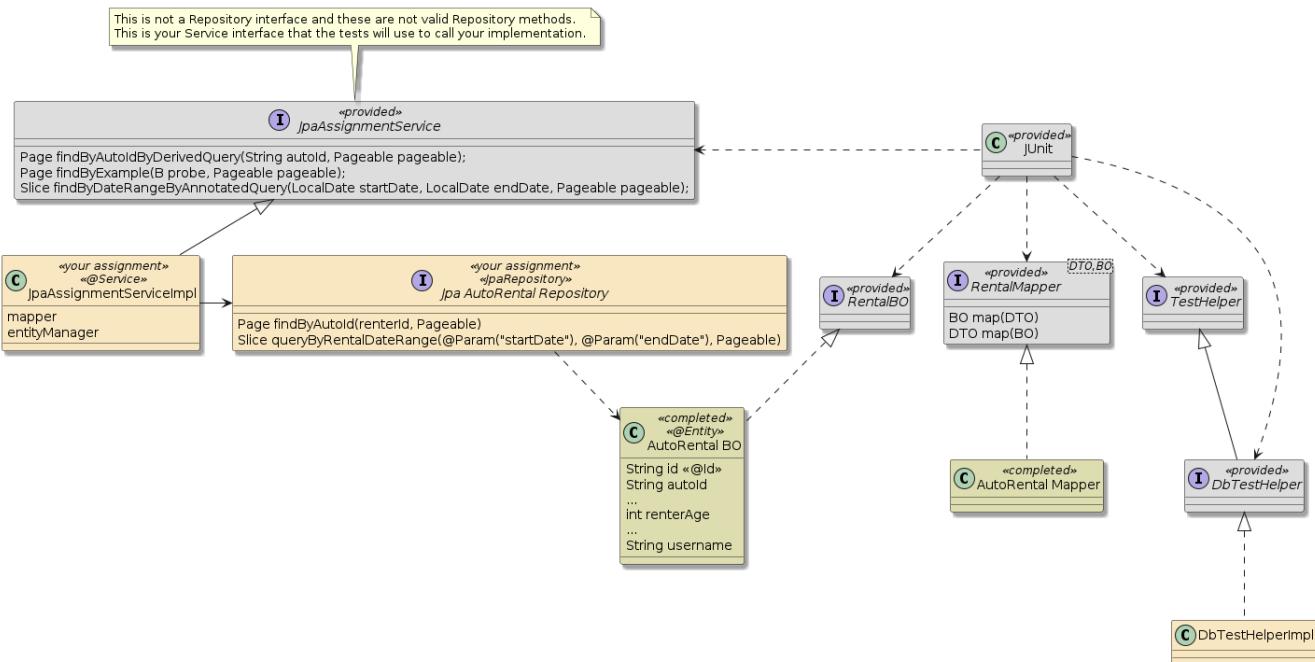


Figure 172. AutoRental Repository

Your work will be focused in the following areas:

- creating a JPA Spring Data Repository for persisting AutoRental BO objects
- implementing repository queries within your `JpaAssignmentService` component

381.3.3. Requirements

1. define a AutoRental JpaRepository that can support basic CRUD and complete the queries defined below.
2. enable JpaRepository use with the `@EnableJpaRepositories` annotation on a `@Configuration` class



Spring Data Repositories are primarily interfaces and the implementation is written for you at runtime using proxies and declarative configuration information.



If your Repository class is not within the default scan path, manually register the package path using the `@EnableJpaRepositories.basePackageClasses` annotation property. This should be done within the `src/main` portion of your code. The JUnit test will make the condition and successful correction obvious.

3. inject the JPA Repository class into your `JpaAssignmentService` component. This will be enough to tell you whether the Repository is properly defined and registered with the Spring context.
4. implement the `findByAutoIdByDerivedQuery` method details which must
 - a. accept a autoId and a Pageable specification with pageNumber, pageSize, and sort specification
 - b. return a Page of matching BOs that comply with the input criteria
 - c. this query must use the [Spring Data Derived Query](#) technique **

5. implement the `findByExample` method details which must
 - a. accept a AutoRental BO probe instance and a Pageable specification with pageNumber, pageSize, and sort specification
 - b. return a Page of matching BOs that comply with the input criteria
 - c. this query must use the [Spring Data Query by Example](#) technique **



Override the default ExampleMatcher to ignore any fields declared with built-in data types that cannot be null.

6. implement the `findByAgeRangeByAnnotatedQuery` method details which must
 - a. accept a minimum and maximum age and a Pageable specification with pageNumber and pageSize
 - b. return a Page of matching BOs that comply with the input criteria and ordered by `id`
 - c. this query must use the [Spring Data Named Query](#) technique and leverage the "AutoRentalBO.findByRenterAgeRange" `@NamedQuery` created in the previous section.



Named Queries do not support adding Sort criteria from the Pageable parameter. An "order by" for `id` must be expressed within the `@NamedQuery`.

```
... order by r.id ASC
```



There is no technical relationship between the name of the service method you are implementing and the repository method defined on the JPA Spring Data Repository. The name of the service method is mangled to describe "how" you must implement it—not what the name of the repository method should be.

7. Enable (and pass) the provided `MyJpa5c_RepositoryTest` that extends `Jpa5c_RepositoryTest`. This test will populate the database with content and issue query requests to your `JpaAssignmentService` implementation.
8. Package the JUnit test case such that it executes with Maven as a surefire test

381.3.4. Grading

Your solution will be evaluated on:

1. declare a `JpaRepository` for an existing JPA `@Entity`
 - a. whether a `JPARepository` was defined and injected into the assignment service helper
2. perform simple CRUD methods using provided repository methods
 - a. whether the database was populated with test instances
3. add paging and sorting to query methods
 - a. whether the query methods were implemented with pageable specifications

4. implement queries based on predicates derived from repository interface methods
 - a. whether a derived query based on method signature was successfully performed
5. implement queries based on POJO examples and configured matchers
 - a. whether a query by example query was successfully performed
6. implement queries based on `@NamedQuery` or `@Query` specification
 - a. whether a query using a `@NamedQuery` or `@Query` source was successfully performed

381.3.5. Additional Details

Chapter 382. Assignment 5b: Spring Data Mongo

382.1. Mongo Client Connection

382.1.1. Purpose

In this portion of the assignment, you will demonstrate your knowledge of setting up a project to work with Mongo. You will:

1. declare project dependencies required for using Spring's MongoOperations/MongoTemplate API
2. define a connection to a MongoDB
3. inject a MongoOperations/MongoTemplate instance to perform actions on a database

382.1.2. Overview

In this portion of the assignment you will be adding required dependencies and configuration properties necessary to communicate with the Flapdoodle test database and an external MongoDB instance.

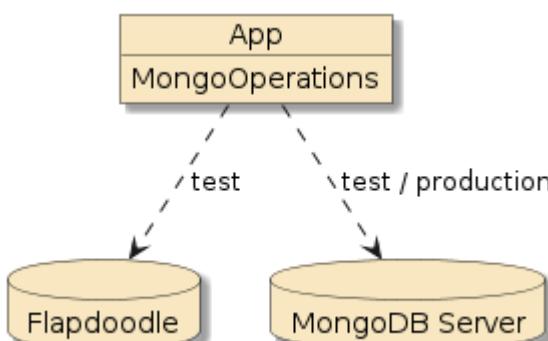


Figure 173. Mongo Client Connection

Postgres access with Docker/Docker Compose

If you have Docker/Docker Compose, you can instantiate a MongoDB instance using the docker-compose scripts in the ejava-springboot root directory.

```
$ docker-compose up -d mongodb  
Creating ejava_mongodb_1 ... done
```



You can also get client access to using the following command.

```
$ docker-compose exec mongodb mongo -u admin -p secret  
...  
Welcome to the MongoDB shell.
```

You can switch between Flapdoodle and Mongo in your tests once you have your property files setup.



```
@SpringBootTest(...)
@ActiveProfiles(profiles={"assignment-tests", "test"}, resolver =
TestProfileResolver.class)
//@ActiveProfiles(profiles={"assignment-tests", "test", "mongodb"})
public class MyMongo5a_ClientTest extends Mongo5a_ClientTest {
```

382.1.3. Requirements

- Configure database properties so that you are able to work with both the Flapdoodle test database and a Mongo instance. Flapdoodle will be good for automated testing. MongoDB will be good for interactive access to the database while developing. Spring Boot will automatically configure tests for Flapdoodle if it is in the classpath and in the absence of a Mongo database URI.

You can turn on verbose MongoDB-related DEBUG logging using the following properties



application.properties

```
logging.level.org.springframework.data.mongodb=DEBUG
```

- provide a `mongodb` profile option to use an external MongoDB server instead of the Flapdoodle test instance

application-mongodb.properties

```
#spring.data.mongodb.host=localhost
#spring.data.mongodb.port=27017
#spring.data.mongodb.database=test
#spring.data.mongodb.authentication-database=admin
#spring.data.mongodb.username=admin
#spring.data.mongodb.password=secret
spring.data.mongodb.uri=mongodb://admin:secret@localhost:27017/test?authSource=admin
```



Configure via Individual Properties or Compound URL

Spring Data Mongo has the capability to set individual configuration properties or via one, compound URL.



This is only a class assignment. Do not store credentials in files checked into

CM or packaged within your Spring Boot executable JAR in a real environment. Make them available via a file location at runtime when outside of a classroom.

Flapdoodle will be Default Database during Testing



Flapdoodle will be the default during testing unless deactivated. I have provided a @Configuration class in the "starter" module, that will deactivate Flapdoodle by the presence of the `spring.data.mongodb` connection properties.

2. Configure the application to establish a connection to the database and establish a MongoOperations (the interface)/MongoTemplate (the commonly referenced implementation class)
 - a. declare a dependency on `spring-boot-starter-data-mongo`
 - b. declare a dependency on the `de.flapdoodle.embed.mongo` database driver for default testing with `scope=test`



See `mongo-book-example pom.xml` for more details on declaring these dependencies.



These dependencies may already be provided through transitive dependencies from the Auto/Renter modules.

3. Enable (and pass) the provided `MyMongo5a_ClientTest` that extends `Mongo5a_ClientTest`. This test will verify connectivity to the database.
4. Package the JUnit test case such that it executes with Maven as a surefire test

382.1.4. Grading

Your solution will be evaluated on:

1. declare project dependencies required for using Spring's MongoOperations/MongoTemplate API
 - a. whether required Maven dependencies were declared to operate and test the application with Mongo
2. define a connection to a MongoDB
 - a. whether a URL to the database was defined when the `mongodb` profile was activated
3. inject an MongoOperations/MongoTemplate instance to perform actions on a database
 - a. whether a MongoOperations client could be injected
 - b. whether the MongoOperations client could successfully communicate with the database

382.1.5. Additional Details

- As with the RDBMS/JPA tests, these MongoDB tests are all client-side DB interaction tests. Calls from JUnit are directed at the service class. The provided starter example supplies an alternate

`@SpringBootConfiguration` test configuration to bypass the extra dependencies defined by the full `@SpringBootApplication` server class—which can cause conflicts. The `@SpringBootConfiguration` class is latched by the "assignment-tests" profile to keep it from being accidentally used by the later API tests.

```
@SpringBootConfiguration
@EnableAutoConfiguration
@Profile("assignment-tests") ①
public class DbAssignmentTestConfiguration {

    @SpringBootTest(classes={DbAssignmentTestConfiguration.class,
        MongoAssignmentDBConfiguration.class,
        DbClientTestConfiguration.class
    })
    @ActiveProfiles(profiles={"assignment-tests", "test"}, resolver =
    TestProfileResolver.class)②
    //@ActiveProfiles(profiles={"assignment-tests", "test", "mongodb"})
    public class MyMongo5a_ClientTest extends Mongo5a_ClientTest {
```

- ① profile prevents `@SpringBootConfiguration` from being used as a `@Configuration` for other tests
- ② `assignment-tests` profile is activated for these service/DB-level tests only

382.2. Mongo Document

382.2.1. Purpose

In this portion of the assignment, you will demonstrate your knowledge of defining a Spring Data Mongo `@Document` class and performing basic CRUD actions. You will:

1. implement basic unit testing using an (seemingly) embedded MongoDB
2. define a `@Document` class to map to MongoDB collection
3. perform whole-document CRUD operations on a `@Document` class using the Java API
4. perform queries with paging properties

382.2.2. Overview

In this portion of the assignment you will be creating a `@Document`/Business Object for a AutoRental, mapping that to a collection, and performing CRUD actions with a `MongoOperations/MongoTemplate`.

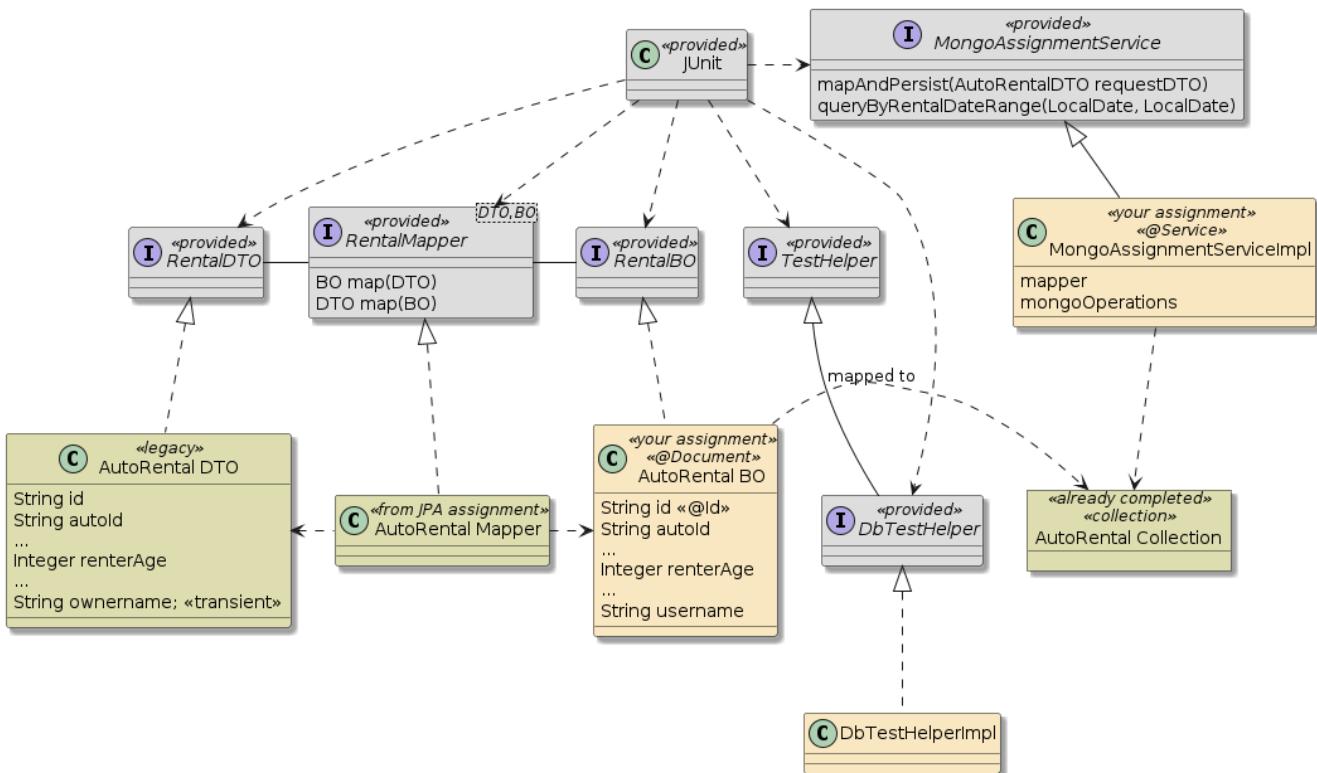


Figure 174. AutoRental Entity

Reuse BO and Mapper classes



It has been designed and expected that you will be able to re-use the same AutoRental BO and Mapper classes from the JPA portion of the assignment. You should not need to create **new** ones. The BO class will need a few Spring Data Mongo annotations but the mapper created for the JPA portion should be 100% reusable here as well.

382.2.3. Requirements

1. Create (*reuse*) a Business Object (BO) class that represents the AutoRental and will be mapped to the database. A RentalBO "marker" interface has been provided for your BO class to implement. It has no properties. All interactions with this object by the JUnit test will be through calls to the testHelper and mapper classes. You must complete the following details:
 - a. identify the class as a Spring Data Mongo **@Document** and map it to the "rentals" collection
 - b. identify a String primary key field with Spring Data Mongo **@Id**
 - c. supply a default constructor
2. Reuse the mapper class from the earlier JPA Entity portion of this assignment.
3. Implement a class that implements the **MongoAssignmentService<DTO, BO>** interface.
 - a. Instantiate this as a component in the "assignment-tests" profile.
4. Implement the **mapAndPersist** method in your **MongoAssignmentService**. It must perform the following:



This is a different **@Id** annotation than the JPA **@Id** annotation.

- a. accept a AutoRental DTO
- b. map the DTO to a AutoRental BO (using your mapper)
- c. save the BO to the database
- d. map the persisted BO to a DTO (will have a primary key assigned)
- e. return the resulting DTO

The BO must be saved. The returned DTO must match the input DTO and express the primary key assigned by the database.

5. Implement the `queryByRentalDateRange` method in your `MongoAssignmentService`. It must perform the following:

- a. query the database to locate matching `AutoRental BO` objects within a `startDate/endDate` range, inclusive



You may use the injected `MongoOperations` client `find` command, a query, and the `AutoRentalBO.class` as a request parameter



You may make use of the following query

```
Query.query(new Criteria().andOperator(
    Criteria.where("startDate").gte(startInclusive),
    Criteria.where("endDate").lte(endInclusive)))
```

- i. `:startInclusive` and `:endInclusive` are variable `LocalDate` values
- ii. order the results by `id` ascending
- b. map the BO list returned from the query to a list of DTOs (using your mapper)
- c. return the list of DTOs
6. Enable (and pass) the provided `MyMongo5b_DocumentTest` that extends `Mongo5b_DocumentTest`. This test will perform checks of the above functionality using:
 - `DbTestHelper`
 - mapper
 - your DTO and BO classes
 - a functional MongoDB environment
7. Package the JUnit test case such that it executes with Maven as a surefire test

382.2.4. Grading

Your solution will be evaluated on:

1. define a `@Document` class to map to MongoDB collection
 - a. whether the BO class was properly mapped to the database, including document and

primary key

2. perform whole-document CRUD operations on a `@Document` class using the Java API
 - a. whether a successful insert and query of the database was performed with the injected `MongoOperations` / `MongoTemplate`

382.2.5. Additional Details

382.3. Mongo Repository

382.3.1. Purpose

In this portion of the assignment, you will demonstrate your knowledge of defining a Mongo Repository. You will:

1. declare a `MongoRepository` for an existing `@Document`
2. implement queries based on predicates derived from repository interface methods
3. implement queries based on POJO examples and configured matchers
4. implement queries based on annotations with JSON query expressions on interface methods
5. add paging and sorting to query methods

382.3.2. Overview

In this portion of the assignment you will define a Mongo Repository to perform basic CRUD and query actions.

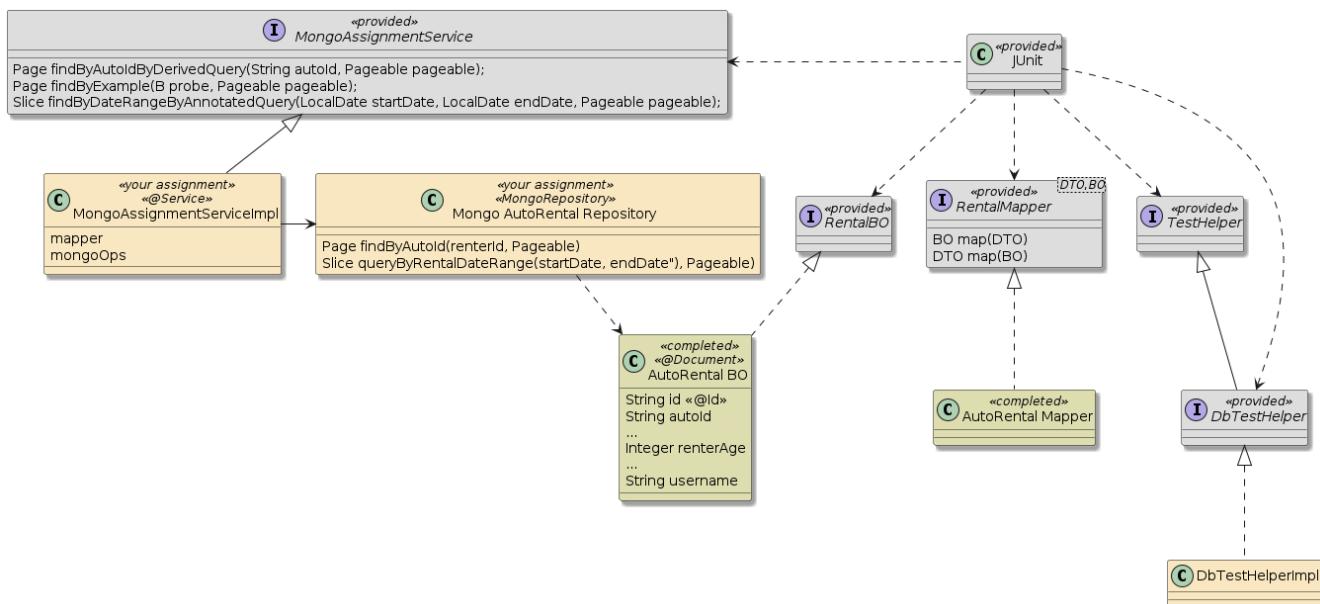


Figure 175. Mongo Repository

Your work will be focused in the following areas:

- creating a Mongo Spring Data Repository for persisting AutoRental BO objects
- implementing repository queries within your `MongoAssignmentService` component

382.3.3. Requirements

1. define a AutoRental MongoRepository that can support basic CRUD and complete the queries defined below.
2. enable MongoRepository use with the `@EnableMongoRepositories` annotation on a `@Configuration` class



Spring Data Repositories are primarily interfaces and the implementation is written for you using proxies and declarative configuration information.



If your Repository class is not within the default scan path, you can manually register the package path using the `@EnableMongoRepositories.basePackageClasses` annotation property. This should be done within a `@Configuration` class in the `src/main` portion of your code. The JUnit test will make the condition and successful correction obvious.

3. inject the Mongo Repository class into the `MongoAssignmentService`. This will be enough to tell you whether the Repository is properly defined and registered with the Spring context.
4. implement the `findByIdByDerivedQuery` method details which must
 - a. accept a `autoId` and a `Pageable` specification with `pageNumber`, `pageSize`, and `sort` specification
 - b. return a `Page` of matching BOs that comply with the input criteria
 - c. this query must use the [Spring Data Derived Query](#) technique **
5. implement the `findByExample` method details which must
 - a. accept a AutoRental BO probe instance and a `Pageable` specification with `pageNumber`, `pageSize`, and `sort` specification
 - b. return a `Page` of matching BOs that comply with the input criteria
 - c. this query must use the [Spring Data Query by Example](#) technique **



Override the default `ExampleMatcher` to ignore any fields declared with built-in data types that cannot be null.

6. implement the `findByDateRangeByAnnotatedQuery` method details which must
 - a. accept a `startDate` and `endDate` (inclusive) and a `Pageable` specification with `pageNumber` and `pageSize`
 - b. return a `Page` of matching BOs that comply with the input criteria and ordered by `id`
 - c. this query must use the [Spring Data JSON-based Query Methods](#) technique and annotate the repository method with a `@Query` definition.



You may use the following JSON query expression for this query. Mongo JSON query expressions only support positional arguments and are zero-relative.

```
value="{ 'startDate':{$lte:?1}, 'endDate':{$gte:?0} }" ①
```

① endDate is position 0 and startDate is position 1 in the method signature

Annotated Queries do not support adding Sort criteria from the Pageable parameter. You may use the following sort expression in the annotation

```
sort="{id:1}"
```



There is no technical relationship between the name of the service method you are implementing and the repository method defined on the Mongo Spring Data Repository. The name of the service method is mangled to describe "how" you must implement it—not what the name of the repository method should be.



7. Enable (and pass) the provided `MyMongo5c_RepositoryTest` that extends `Mongo5c_RepositoryTest`. This test will populate the database with content and issue query requests to your `MongoAssignmentService` implementation.
8. Package the JUnit test case such that it executes with Maven as a surefire test

382.3.4. Grading

Your solution will be evaluated on:

1. declare a `MongoRepository` for an existing `@Document`
 - a. whether a MongoRepository was declared and successfully integrated into the test case
2. implement queries based on predicates derived from repository interface methods
 - a. whether a dynamic query was implemented via the expression of the repository interface method name
3. implement queries based on POJO examples and configured matchers
 - a. whether a query was successfully implemented using an example with a probe document and matching rules
4. implement queries based on annotations with JSON query expressions on interface methods
 - a. whether a query was successfully implemented using annotated repository methods containing a JSON query and sort documents
5. add paging and sorting to query methods
 - a. whether queries were performed with sorting and paging

382.3.5. Additional Details

Chapter 383. Assignment 5c: Spring Data Application

383.1. API/Service/DB End-to-End

383.1.1. Purpose

In this portion of the assignment, you will demonstrate your knowledge of integrating a Spring Data Repository into an end-to-end application, accessed through an API. You will:

1. implement a service tier that completes useful actions
2. implement controller/service layer interactions relative to DTO and BO classes
3. determine the correct transaction propagation property for a service tier method
4. implement paging requests through the API
5. implement page responses through the API

383.1.2. Overview

In this portion of the assignment you will be taking elements of the application that you have worked on and integrate them into an end-to-end application from the API, thru services, security, the repository, and to the database and back.

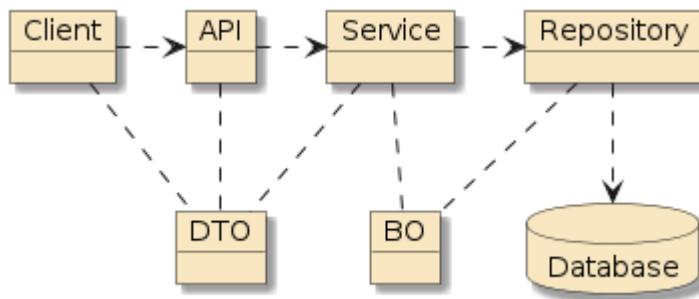


Figure 176. API/Service/DB End-to-End

The fundamental scope of the assignment is to perform existing AutoRentals use cases (including security layer(s)) but updated with database and the impacts of database interaction related to eventual size and scale. You will:

- chose either a RDBMS/JPA or Mongo target solution
- replace your existing AutoRental Service and Repository implementation classes with implementations that are based upon your selected repository technology

For those that



- augmented your assignment2/API solution in-place to meet the requirements of each follow-on assignment

- you will continue that pattern by replacing the core service logic with mapper/BO/repository logic.
- layered your solution along assignment boundaries
 - you will override the assignment2 service and repository components with a new service implementation based on the mapper/BO/repository logic. This layer will be inserted into your assignment3 SecurityWrapper. The support modules show an example of doing this.
- update the controller and service interfaces to address paging

It is again, your option of whether to

- simply add the new paging endpoint to your existing controller and API client class
- subclass the controller and API class to add the functionality



The Autos/Renters support modules are provided in a layered approach to help identify what is new with each level and to keep what you are basing your solutions on consistent. It is much harder to implement the layered approach, but offers some challenges and experience in integrating multiple components.

- leave the Autos and Renters implementation as in-memory repositories

There are two optional support modules supplied:



- `autorenters-support-svcjpa` provides an implementation of Autos using `JpaRepository`
- `autorenters-support-svcmongo` provides an implementation of Renters using `MongoRepository`

The tests within each module work but extensive testing with AutoRentals has not been performed. It is anticipated that you will continue to use the in-memory Autos and Renters that you have been using to date. However, it is your option to use those modules in any way.

Continued use of in-memory Autos and Renters is expected



The `autorenters-support-svcjpa` and `autorenters-support-svcmongo` modules are provided as examples of how the flow can be implemented. It is not a requirement that you change from the in-memory versions for Autos and Renters to complete this assignment.

383.1.3. Requirements

1. Select a database implementation choice (`JpaRepository` or `MongoRepository`).



This is a choice to move forward with. The option you don't select will still be

part of your dependencies, source tree, and completed unit integration tests.

2. Update/Replace your legacy AutoRental Service and Repository components with a service and repository based on Spring Data Repository.

- a. all CRUD calls will be handled by the Repository—no need for DataSource, EntityManager or MongoOperations/MongoTemplate
- b. all end-to-end queries must accept Pageable and return Page



By following the rule early in assignment 2, you made this transition extremely easy on yourself.

- c. the service should

- i. accept DTO types as input and map to BOs for interaction with the Repository



This should be consistent with your original service interface. The only change should be the conversion of DTO to BO and back.



It is acceptable to retain much of your legacy service logic (e.g., validation) working primarily with DTOs. However, you will need to convert them to BOs to interact with the database.

- i. map BO types to DTOs as output



This should also be consistent with your original service interface. The [Page class has a nice/easy way to map between Page<T1> to Page<T2>](#). When you combine that with your mapper—it can be a simple one-line of code.

```
dtos = bos.map(bo->map(bo));
```

3. Add the capability to your controller to accept full paging parameters. For those augmenting in-place, you may simply modify your existing finder methods to accept/return the additional information. For those adopting the layered approach, you may add an additional URI to accept/return the additional information.

Example new Paged URI for Layered Approach



```
public interface HomesPageableAPI extends HomesAPI {  
    public static final String HOMES_PAGED_PATH = "/api/homes/paged";  
};
```



There is a convenience method within [PageableDTO](#) (from [ejava-dto-util](#)) that will convert pageNumber, pageSize, and sort to a Spring Data [Pageable](#).

```
@GetMapping(path = HomesPageableAPI.HOMES_PAGED_PATH, ...)
```

```

public ResponseEntity<HomePageDTO> getHomesPage(
    @RequestParam(value = "pageNumber", required = false) Integer pageNumber,
    @RequestParam(value = "pageSize", required = false) Integer pageSize,
    @RequestParam(value = "sort", required = false) String sort) {
    Pageable pageable = PageableDTO.of(pageNumber, pageSize, sort)
        .toPageable();
}

```

- Add the capability to your controller to return paging information with the contents. The current pageNumber, pageSize, and sort that relate to the supplied data must be returned with the contents.

You may use the `PageDTO<T>` class (from `ejava-dto-util`) to automate/encapsulate much of this. The primary requirement is to convey the information. The option is yours of whether to use this library demonstrated in the class JPA and Mongo examples as well as the Autos and Renters examples (from `autorenters-support-pageable-svc`)



```

public class HomePageDTO extends PageDTO<HomeDTO> {
    protected HomePageDTO() {}
    public HomePageDTO(Page<HomeDTO> page) {
        super(page);
    }
}

```

```

Page<HomeDTO> result = service.getHomes(pageable);
HomePageDTO resultDTO = new HomePageDTO(result);
return ResponseEntity.ok(resultDTO);

```

- Add the capability to your API calls to provide and process the additional page information.

There is a convenience method within `PageableDTO` (from `ejava-dto-util`) that will serialize the pageNumber, pageSize, and sort of Spring Data's Pageable into query parameters.



```

PageableDTO pageableDTO = PageableDTO.fromPageable(pageable); ①
URI url = UriComponentsBuilder
    .fromUri(homesUrl)
    .queryParams(pageableDTO.getQueryParams()) ②
    .build().toUri();

```

① create DTO abstraction from Spring Data's local Pageable abstraction

② transfer DTO representation into query parameters

6. Write a JUnit Integration Test Case that will

- populate the database with multiple AutoRentals with different and similar properties
- query for AutoRentals based on a criteria that will match some of the AutoRentals and return a page of contents that is less than the total matches in the database. (i.e., make the pageSize smaller than total number of matches)
- page through the results until the end of data is encountered



Skeletal XxxPagingNTest tests are provided within the starter module to form the basis of these tests. You are to implement either the Jpa or Mongo version.



Again — the DTO Paging framework in common and the JPA Songs and Mongo Books examples should make this less heroic than it may sound.



The requirement is not that you integrate with the provided DTO Paging framework. The requirement is that you implement end-to-end paging and the provided framework can take a lot of the API burden off of you. You may implement page specification and page results in a unique manner as long as it is end-to-end.

7. Package the JUnit test case such that it executes with Maven as a surefire test

383.1.4. Grading

Your solution will be evaluated on:

1. implement a service tier that completes useful actions
 - a. whether you successfully implemented a query for AutoRentals for a specific autoId
 - b. whether the service tier implemented the required query with Pageable inputs and a Page response
 - c. whether this was demonstrated thru a JUnit test
2. implement controller/service layer interactions when it comes to using DTO and BO classes
 - a. whether the controller worked exclusively with DTO business classes and implemented a thin API facade
 - b. whether the service layer mapped DTO and BO business classes and encapsulated the details of the service
3. determine the correct transaction propagation property for a service tier method
 - a. depending on the technology you select and the usecase you have implemented — whether the state of the database can ever reach an inconsistent state
4. implement paging requests through the API
 - a. whether your controller implemented a means to express Pageable request parameters for queries

5. implement page responses through the API

- a. whether your controller supported returning a page-worth of results for query results

383.1.5. Additional Details

1. There is a set of tests under the "regression" folder that can be enabled. These will run a series of tests ported forward from the API and Security modules to better verify your implementation logic is sound. You must set:

- rentals.impl=jpa (the default) to activate the JPA configuration
- rentals.impl=mongo to activate the Mongo configuration

Service Configuration Defaults to JPA

```
//rentals.impl=jpa
@ConditionalOnProperty(prefix = "rentals", name="impl", havingValue = "jpa",
matchIfMissing = true)
...
public class JpaAutoRentalsConfiguration {

//rentals.impl=mongo
@ConditionalOnProperty(prefix = "rentals", name="impl", havingValue = "mongo",
matchIfMissing = false)
...
public class MongoAutoRentalsConfiguration {
```

Bean Validation

copyright © 2024 jim stafford (jim.stafford@jhu.edu)

Chapter 384. Introduction

Well-designed software components should always be designed according to a contract of what is required of inputs and outputs; constraints; or pre-conditions and post-conditions. Validation of inputs and outputs need to be performed at component boundaries. These conditions need to be well-advertised, but ideally the checking of these conditions should not overwhelm the functional aspects of the code.

Manual Validation

```
public PersonPocDTO createPOC(PersonPocDTO personDTO) {  
    if (null == personDTO) {  
        throw new BadRequestException("createPOC.person: must not be null");  
    } else if (StringUtils.isNotBlank(personDTO.getId())) {  
        throw new InvalidInputException("createPOC.person.id: must be null");  
    } ... ①
```

① business logic is possibly overwhelmed by validation concerns and actual checks

This lecture will introduce working with the Bean Validation API to implement declarative and expressive validation.

Declarative Bean Validation API

```
@Validated(PocValidationGroups.CreatePlusDefault.class)  
public PersonPocDTO createPOC(  
    @NotNull  
    @Valid PersonPocDTO personDTO); ①
```

① conditions well-advertised and isolated from target business logic

384.1. Goals

The student will learn:

- to add declarative pre-conditions and post-conditions to components using the Bean Validation API
- to define declarative validation constraints
- to implement custom validation constraints
- to enable injected call validation for components
- to identify patterns/anti-patterns for validation

384.2. Objectives

At the conclusion of this lecture and related exercises, the student will be able to:

1. add Bean Validation to their project

2. add declarative data validation constraints to types and method parameters
3. configure a `ValidatorFactory` and obtain a `Validator`
4. programmatically validate an object
5. programmatically validate parameters to and response from a method call
6. inspect constraint violations
7. enable Spring/AOP validation for components
8. implement a custom validation constraint
9. implement a cross-parameter validation constraint
10. configure Web API constraint violation responses
11. configure Web API parameter validation
12. configure JPA validation
13. configure Spring Data Mongo Validation
14. identify some patterns/anti-patterns for using validation

Chapter 385. Background

[Bean Validation](#) is a standard that originally came out of Java EE/SE as JSR-303 (1.0) in the 2009 timeframe and later updated with JSR-349 (1.1) in 2013, [JSR-380 \(2.0\)](#) in 2017, and finally [3.0](#) in 2020 with the Java package and XML namespace changes from javax to jakarta.

It was meant to simplify validation—reducing the chance of error and to reduce the clutter of validation within the business code that required validation. The standard is not specific any particular tier (e.g., UI, Web, Service, DB) but has been integrated into several of the individual frameworks. ^[1]

Implementations include:

- [Hibernate Validator](#)
- [Apache BVal](#)

Hibernate Validator was the original and current reference implementation and used within Spring Boot today.

[1] ["Jakarta Bean Validation specification"](#), Gunnar Morling, 2020

Chapter 386. Dependencies

To get started with validation in Spring Boot—we add a dependency on [spring-boot-starter-validation](#).

Validation Dependency

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

That will bring in the validation reference implementation from Hibernate and an implementation for regular expression validation constraints.

Validation Transient Dependencies

```
[INFO] +- org.springframework.boot:spring-boot-starter-validation:jar:3.3.2:compile
[INFO] |   +- org.apache.tomcat.embed:tomcat-embed-el:jar:10.1.26:compile ③
[INFO] |   \- org.hibernate.validator:hibernate-validator:jar:8.0.1.Final:compile ②
[INFO] |       +- jakarta.validation:jakarta.validation-api:jar:3.0.2:compile ①
[INFO] |       +- org.jboss.logging:jboss-logging:jar:3.5.3.Final:compile
[INFO] |       \- com.fasterxml:classmate:jar:1.7.0:compile
```

① overall Bean Validation API

② Bean Validation API reference implementation from Hibernate

③ regular expression implementation for regular expression constraints

Chapter 387. Declarative Constraints

At the core of the Bean Validation API are declarative constraint annotations we can place directly into the source code.

387.1. Data Constraints

The following snippet shows a class with a property that is required to be not-null when valid.

Java Class with Validation Constraint Annotation(s)

```
import jakarta.validation.constraints.NotNull;  
...  
class AClass {  
    @NotNull  
    private String aValue;  
    ...
```

Constraints do not Actively Prevent Invalid State



The constraint does not actively prevent the property from being set to an invalid value. Unlike with the Lombok annotations, no class code is written as a result of the validation annotations. The constraint will identify whether the property is currently valid when validated. The validating caller can decide what to do with the invalid state.

387.2. Common Built-in Constraints

You can find a list of built-in constraints in the [Bean Validation Spec](#) and the [Hibernate Validator documentation](#). A few common ones include:

Table 26. Common Built-in Validator Constraints

- | | |
|---|---|
| <ul style="list-style-type: none">• @Null, @NotNull• @NotBlank, @NotEmpty• @Past, @Future• @Min, Max - collection size | <ul style="list-style-type: none">• @Size(min, max) - value limit• @Positive, @Negative• @PositiveOrZero, @NegativeOrZero• @Pattern(regex) |
|---|---|

Additional constraints are provided by:

- [Hibernate Additional Constraints](#) (e.g., @CreditCardNumber)
- [Java Bean Validation Extension \(JBVExt\)](#) (e.g. @Alpha, @IsDate, @IPv4)

We will take a look at how to create a custom constraint later in this lecture.

387.3. Method Constraints

We can provide pre-condition and post-condition constraints on methods and constructors.

The following snippet shows a method that requires a non-null and valid parameter and will return a non-null result. Constraints for input are placed on the individual input parameters. Constraints on the output (as well as cross-parameter constraints) are placed on the method. The `@Validated` annotation is added to components to trigger Spring to enable validation for injected components.

Java Method Declaration with Parameter

```
import jakarta.validation.Valid;
import org.springframework.validation.annotation.Validated;
...
@Component
@Validated ③
public class AService {
    @NotNull ②
    public String aMethod(@NotNull @Valid AClass aParameter) { ①
        return ...;
    }
}
```

① method requires a non-null parameter with valid content

② the result of the method is required to be non-null

③ `@Validated` triggers Spring's use of the Bean Validation API to validate the call



Null Properties Are Considered @Valid Unless Explicitly Constrained with @NotNull

It is a best practice to consider a null value as valid unless explicitly constrained with `@NotNull`.

We will eventually show all this integrated within Spring, but first we want to make sure we understand the plumbing and what Spring is doing under the covers.

Chapter 388. Programmatic Validation

To work with the Bean Validation API directly, our initial goal is to obtain a standard `jakarta.validation.Validator` instance.

Programmatic Validation Requires Validator

```
import jakarta.validation.Validator;  
...  
Validator validator;
```

This can be obtained manually or through injection.

388.1. Manual Validator Instantiation

We can get a Validator using one of the `Validation` builder methods to return a `ValidatorFactory`.

The following snippet shows the builder providing an instance of the default factory provider, with the default configuration. If we have no configuration changes, we can simplify with a call to `buildDefaultValidatorFactory()`. The `Validator` instance is obtained from the `ValidatorFactory`. Both the factory and validator instances are thread-safe. We will come back to the `configure()` method options later.

Standard jakarta.validation.Validator Interface

```
import jakarta.validation.Validation;  
...  
ValidatorFactory myValidatorFactory = Validation.byDefaultProvider()  
    .configure()  
    //configuration commands  
    .buildValidatorFactory(); ①  
//ValidatorFactory myValidatorFactory = Validation.buildDefaultValidatorFactory();  
Validator myValidator = myValidatorFactory.getValidator(); ①
```

① factory and validator instances are thread-safe, initialized during bean construction, and used during instance methods

388.2. Inject Validator Instance

With the validation starter dependency comes a default Validator. For components, we can simply have it injected.

Injecting Validator

```
@Autowired  
private Validator validator;
```

388.3. Customizing Injected Instance

If we want the best of both worlds and have some customizations to make, we can define a `@Bean` factory to replace the AutoConfigure and return our version of the `Validator` instead.

Custom Validator @Bean Factory

```
@Bean ①
public Validator validator() {
    return Validation.byDefaultProvider()
        .configure()
        //configuration commands
        .buildValidatorFactory()
        .getValidator();
}
```

① A custom `Validator @Bean` within the application will override the default provided by Spring Boot

388.4. Review: Class with Constraint

The following validation example(s) will use the following class with a non-null constraint on one of its properties.

Example Class with Constraint

```
@Getter
public class AClass {
    @NotNull
    private String aValue;
    ...
}
```

388.5. Validate Object

The most straight forward use of the validation programmatic API is to validate individual objects. The object to be validated is supplied and a `Set<ConstraintViolation>` is returned. No exceptions are thrown by the Bean Validation API itself for constraint violations. Exceptions are only thrown for invalid use of the API and to report violations within frameworks like [Contexts and Dependency Injection \(CDI\)](#) or Spring Boot.

The following snippet shows an example of using the validator to validate an object with at least one constraint error.

Validate Object

```
//given - @NotNull aProperty set to null by ctor
AClass invalidAClass = new AClass();
//when - checking constraints
```

```

Set<ConstraintViolation<AClass>> violations = myValidator.validate(invalidAClass); ①
violations.forEach(v-> log.info("field name={}, value={}, violated={}",
    v.getPropertyPath(), v.getInvalidValue(), v.getMessage()));
//then - there will be at least one violation
then(violations).isNotEmpty(); ②

```

① programmatic call to validate object

② non-empty return set means violations were found

The result of the validation is a `Set<ConstraintViolation>`. Each constraint violation identifies the:

- path to the field in error
- an error message
- invalid value
- descriptors for the annotation and validator

The following shows the output of the example.

Validate Object Text Output

```
field name=aValue, value=null, violated=must not be null
```

Specific Property Validation

We can also validate a value against the definition of a specific property



- `validateProperty(T object, propertyName, groups)`
- `validateValue(Class<T> beanType, propertyName, value, groups)`

388.6. Validate Method Calls

We can also validate calls to and results from methods (and constructors too). This is commonly performed by AOP code—rather than anything closely related to the business logic.

The following snippet shows a class with a method that has input and response constraints. The input parameter must be valid and not null. The response must also be not null. A `@Valid` constraint on an input argument or response will trigger the object validation—which we just demonstrated—to be performed.

Validate Method Calls

```

public class AService {
    @NotNull
    public String aMethod(@NotNull @Valid AClass aParameter) { ① ②
        return ...
    }
}

```

① `@NotNull` constrains `aParameter` to always be non-null

② `@Valid` triggers validation contents of `aParameter`

With those validation rules in place, we can check them for the following sample call.

Get Reference to Target Service and Input Parameters

```
//given
AService myService = new AService(); ①
AClass myValue = new AClass();
//when
String result = myService.aMethod(myValue);
```

- ① Note: Service shown here as POJO. Must be injected for container to intercept and subject to validation



Please note that the code above is a plain POJO call. Validation is only automatically performed for injected components. We will use this call to describe how to programmatically validate a method call.

388.7. Identify Method Using Java Reflection

Before we can validate anything, we must identify the descriptors of the call and resolve a **Method** reference using Java Reflection.

In the following example snippet we locate the method called **aMethod** on the **AService** class that accepts one parameter of **AClass** type.

Identify Method to Call using Java Reflection

```
Object[] methodParams = new Object[]{ myValue };
Class<?>[] methodParamTypes = new Class<?>[]{ AClass.class };
Method methodToCall = AService.class.getMethod("aMethod", methodParamTypes);
```

The code above has now resolved a reference to the following method call.

Resolved Method Reference

```
(AService)myService).aMethod((AClass)myValue);
```

388.8. Programmatically Check for Parameter Violations

Without actually making the call, we can check whether the given parameters violate defined method constraints by accessing the **ExecutableValidator** from the **Validator** object. **Executable** is a generalized **java.lang.reflect** type for **Method** and **Constructor**.

Programmatically Check For Parameter Violations

```
//when
```

```
Set<ConstraintViolation<AService>> violations = validator
    .forExecutables() ①
    .validateParameters(myService, methodToCall, methodParams);
```

① returns `ExecutableValidator`

The following snippet shows the reporting of the validation results when subjecting our `myValue` parameter to the defined validation rules of the `aMethod()` method.

Invalid Input is Identified

```
//then
then(violations).hasSize(1);
ConstraintViolation<?> violation = violations.iterator().next();
then(violation.getPropertyPath().toString()).isEqualTo("aMethod.arg0.aValue");
then(violation.getMessage()).isEqualTo("must not be null");
then(violation.getInvalidValue()).isEqualTo(null);
then(violation.getInvalidValue()).isEqualTo(myValue.getAValue());
```

388.9. Validate Method Results

We can also validate what is returned against the defined rules of the `aMethod()` method using the same service instance and method reflection references from the parameter validation. Except in this case, `methodToCall` has already been called, and we are now holding onto the result value.

The following example shows an example of validating a null result against the return rules of the `aMethod()` method.

Validating Value Relative to Method Return Value Constraints

```
//given
String nullResult = null;
//when
violations = validator.forExecutables()
    .validateReturnValue(myService, methodToCall, nullResult);
```

Since null is not allowed, one violation is reported.

Method Return Value Constraint Violation(s)

```
//then
then(violations).hasSize(1);
violation = violations.iterator().next();
then(violation.getPropertyPath().toString()).isEqualTo("aMethod.<return value>");
then(violation.getMessage()).isEqualTo("must not be null");
then(violation.getInvalidValue()).isEqualTo(nullResult);
```

Chapter 389. Method Parameter Naming

Validation is able to easily gather meaningful field path information from classes and properties. When we validated the `AClass` instance, we were told the given name of the property in error supplied from reflection.

Java Class with Property

```
class AClass {  
    @NotNull  
    private String aValue;  
    ...
```

Validation Result using Property Name

```
field name=aValue, value=null, violated=must not be null
```

However, reflection by default does not provide the given names of parameters — only the position.

Java Method Declaration with Parameter

```
public class AService {  
    @NotNull  
    public String aMethod(@NotNull @Valid AClass aParameter) {  
        return ...  
    }
```

Validation result using Argument Position

```
[ERROR]  SelfDeclaredValidatorTest.method_arguments:96  
expected: "aMethod.aParameter.aValue"  
but was: "aMethod.arg0.aValue" ①
```

① By default, argument position supplied (`arg0`) — not argument name

There are two ways to solve this.

389.1. Option 1: Add -parameters to Java Compiler Command

The first way to solve this would be to add the `-parameters` option to the Java compiler.

The following snippet shows how to do this for the `maven-compiler-plugin`. Note that this only applies to what is compiled with Maven and not what is actively worked on within the IDE.

Add -parameters to Maven Compile Command

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <configuration>
    <parameters>true</parameters>
  </configuration>
</plugin>
```



The above appears to work fine with the [Maven compiler plugin](#) 3.10.1, but I encountered issues getting that working with the older 3.8.1 (without an explicit `-parameters` in `compilerArgs`).

389.2. Option 2: Add Custom ParameterNameProvider

Another way to help provide parameter names is to configure the `ValidatorFactory` with a `ParameterNameProvider`.

Add Custom ParameterNameProvider

```
ValidatorFactory myValidatorFactory = Validation.byDefaultProvider()
  .configure()
  .parameterNameProvider(new MyParameterNameProvider()) ①
  .buildValidatorFactory();
```

① configuring `ValidatorFactory` with custom parameter name provider

389.2.1. ParameterNameProvider

The following snippets show the skeletal structure of a sample `ParameterNameProvider`. It has separate incoming calls for `Method` and `Constructor` calls and must produce a list of names to use. This particular example is simply returning the default. Example work will be supplied next.

Custom Parameter Name Provider Skeletal Shell

```
import jakarta.validation.ParameterNameProvider;
import java.lang.reflect.Constructor;
import java.lang.reflect.Executable;
import java.lang.reflect.Method;
import java.lang.reflect.Parameter;
...
public class MyParameterNameProvider implements ParameterNameProvider {
  @Override
  public List<String> getParameterNames(Constructor<?> ctor) {
    return getParameterNames((Executable) ctor);
  }
  @Override
  public List<String> getParameterNames(Method method) {
```

```

        return getParameterNames((Executable) method);
    }
    protected List<String> getParameterNames(Executable method) {
        List<String> argNames = new ArrayList<>(method.getParameterCount());
        for (Parameter p: method.getParameters()) {
            //do something to determine Parameter p's desired name
            String argName=p.getName(); ①
            argNames.add(argName);
        }
        return argNames; ②
    }
}

```

① real work to determine the parameter name goes here

② must return a list of parameter names of expected size

389.2.2. Named Parameters

The Bean Validation API does not provide a way to annotate parameters with names. They left that up to us and other Java standards. In this example, I am making use of `jakarta.inject.Named` to supply a textual name of my choice.

Java Method Declaration with @Named Parameter

```

import jakarta.inject.Named;
...
private static class AService {
    @NotNull
    public String aMethod(@NotNull @Valid @Named("aParameter") AClass aParameter) {①
        return ...
    }
}

```

① `@Named` annotation is providing a name to use for `MyParameterNameProvider`

389.2.3. Determining Parameter Name

Now we can update `MyParameterNameProvider` to look for and use the `@Named.value` property if provided or default to the name from reflection.

Processing @Named Annotation to Obtain Parameter Name

```

Named named = p.getAnnotation(Named.class);
String argName=named!=null && StringUtils.isNotBlank(named.value()) ?
    named.value() : //@Named.property
    p.getName();           //default reflection name
argNames.add(argName);

```

The result is a property path that possibly has more meaning.

Before/After Parameter Naming Solution(s) Applied

original: aMethod.arg0 aValue

assisted: aMethod.aParameter.aValue

Chapter 390. Graphs

Constraint validation can follow a graph of references annotated with `@Valid`.

The following snippet shows an example set of parent classes — each with a reference to equivalent child instances. The child instance will be invalid in both cases. Only the `TraversingParent` will be considered invalid because only that parent class defines a requirement that the child be valid (using `@Valid`).

Validation Graph Example Classes

```
class Child {  
    @NotNull  
    String cannotBeNull; ①  
}  
class NonTraversingParent {  
    Child child = new Child(); ②  
}  
class TraversingParent {  
    @Valid ④  
    Child child = new Child(); ③  
}
```

- ① child attribute constrained to not be null
- ② child instantiated with default instance but not annotated
- ③ child instantiated with default instance
- ④ annotation instructs validator to traverse to the child and validate

390.1. Graph Non-Traversal

We know from the previous chapter that we can validate any constraints on an object by passing the instance to the `validate()` method. However, validation will stop there if there are no `@Valid` annotations on references.

The following snippet shows an example of a parent with an invalid child, but due to the lack of `@Valid` annotation, the child state is not evaluated with the parent.

No Validation Traversal to Child

```
//given  
Object nonTraversing = new NonTraversingParent(); ①  
//when  
Set<ConstraintViolation<Object>> violations = validator.validate(nonTraversing); ②  
//then  
then(violations).isEmpty(); ③
```

- ① parent contains an invalid child

- ② constraint validation does not traverse from parent to child
- ③ child errors are not reported because they were never checked

390.2. Graph Traversal

Adding the `@Valid` annotation to an object reference activates traversal to and validation of the child instance. This can be continued to grandchildren with follow-on child `@Valid` annotations.

The following snippet shows an example of a parent whose validation traverses to the child because of the `@Valid` annotation.

Validation Traversal to Child

```
import jakarta.validation.Valid;  
...  
//given  
Object traversing = new TraversingParent(); ①  
//when  
Set<ConstraintViolation<Object>> violations = validator.validate(traversing); ②  
//then  
String errorMsgs = violations.stream()  
    .map(v->v.getPropertyPath().toString()+ ":" + v.getMessage())  
    .collect(Collectors.joining("\n"));  
then(errorMsgs).contains("child.cannotBeNull:must not be null"); ③  
then(violations).hasSize(1);
```

- ① parent contains an invalid child
- ② constraint validation traverses relationship and performed on parent and child
- ③ child errors reported

Chapter 391. Groups

The Bean Validation API supports validation within different contexts using groups. This allows us to write constraints for specific situations, use them when appropriate, and bypass them when not pertinent. The earlier examples all used the default `jakarta.validation.groups.Default` group and were evaluated by default because no group was specified in the call to `validate()`.

We can define our own custom groups using Java interfaces.

391.1. Custom Validation Groups

The following snippet shows an example of two groups. `Create` should only be applied during creation. `CreatePlusDefault` should only be applied during creation but will also apply default validation. `UpdatePlusDefault` can be used to denote constraints unique to updates.

Custom Validation Groups

```
import jakarta.validation.groups.Default;  
...  
public interface PocValidationGroups { ③  
    interface Create{} ①  
    interface CreatePlusDefault extends Create, Default{} ②  
    interface UpdatePlusDefault extends Default{}
```

① custom group name to be used during create

② groups that extend another group have constraints for that group applied as well

③ outer interface is not required, Used in example to make the purpose and source of the group obvious

Each interface is used in two ways:

1. **on the constraint** to denote the group it applies to. Any constraint annotated with a group (`Create`) will get validated if validation is triggered for that same type (`Create`) or subtype (`CreatePlusDefault`).
2. denote the **constraint types to validate**. The constraint types validated will be of the same type or a sub-type.

391.2. Applying Groups

We can assign specific groups to a constraint. In the following example,

- `@Null id` will only be validated when validating the `Create` or `CreatePlusDefault` groups
- `@Past dob` will be validated for both `CreatePlusDefault` and `Default` validation
- `@Size contactPoints` and `@NotNull contactPoints` will each be validated the same as `@Past dob`. The default group is `Default` when left unspecified.

```

public class PersonPocDTO {
    @Null(groups = PocValidationGroups.Create.class, ①
        message = "cannot be specified for create")
    private String id;
    private String firstName;
    private String lastName;
    @Past(groups = Default.class) ②
    private LocalDate dob;
    @Size(min=1, message = "must have at least one contact point") ③
    private List<@NotNull @Valid ContactPointDTO> contactPoints;

```

① explicitly setting group to `Create`, which does not include `Default`

② explicitly setting group to `Default`

③ implicitly setting group to `Default`

391.3. Skipping Groups

With use-case-specific groups assigned, we can have certain defined constraints ignored.

The following example shows the validation of an object. It has an assigned `id`, which would make it invalid for a create. However, there are no violations reported because the group for the `@Null` `id` constraint was not validated because its assigned group (`Create`) is not a sub-type of `Default`.

Validation Group Skipped Example

```

//given
ContactPointDTO invalidForCreate = contactDTOFactory.make(ContactDTOFactory.oneUpId);
①
//when
Set<ConstraintViolation<ContactPointDTO>> violations = validator.validate
(invalidForCreate); ②
//then
then(violations).hasSize(0);

```

① object contains non-null `id`, which is invalid for create scenarios

② implicitly validating against the default group. `Create` group constraints not validated

Can Redefine Default Group for Type with `@GroupSequence`



The Bean Validation API makes it possible to `redefined the default group` for a particular type using a `@GroupSequence`.

391.4. Applying Groups

To apply a non-default group to the validation—we can simply add their interface identifiers in a sequence after the object passed to `validate()`.

The following snippet shows an example of the `CreatePlusDefault` group being applied. The `@Null` `id`

constraint is validated and reported in error because the group it was assigned to was part of the validation command.

Validation Group Applied Example

```
...
//when
violations = validator.validate(invalidForCreate, CreatePlusDefault.class); ①
//then
then(violations).hasSize(1); ②
then(errors(violations)).contains("id:cannot be specified for create"); ③
```

① validating both the `CreatePlusDefault` and `Default` groups

② `@Null id` violation detected and reported

③ `errors()` is a local helper method written to extract field and text from violation

Chapter 392. Multiple Groups

We have two ways of treating multiple groups

validate all

performed by passing more than one group to the `validate()` method. Each group is validated in a non-deterministic manner

short circuit

performed by defining a `@GroupSequence`. Each group is validated in order, and the sequence is short-circuited when there is a failure.

392.1. Example Class with Different Groups

The following snippet shows an example of a class with validations that perform at different costs.

- `@Size email` is thought to be straightforward to validate
- `@Email email` is thought to be a more detailed validation
- the remaining validations have not been included in this class hierarchy

Class with Different Groups

```
public class ContactPointDTO {  
    @Null(groups = {Create.class},  
          message = "cannot be specified for create")  
    private String id;  
    @NotNull  
    private String name;  
    @Size(min=7, max=40, groups= SimplePlusDefault.class) ①  
    @Email(groups = DetailedOnly.class) ②  
    private String email;
```

① `@Size email` is thought to be a cheap sanity check, but overly simplistic

② `@Email email` is thought to be thorough validation, but only worth it for reasonably sane values

The following snippet shows the groups being used in this example.

Example Groups

```
interface Create{}  
interface SimplePlusDefault extends Default {}  
interface DetailedOnly {}
```

392.2. Validate All Supplied Groups

When groups are passed to validate in a sequence, all groups in that sequence are validated.

The following snippet shows an example with `SimplePlusDefault` and `DetailedOnly` supplied to `validate()`. Each group will be validated, no matter what the results are.

Validate All Supplied Groups Example

```
String nameErr="name:must not be null"; //Default Group
String sizeErr="email:size must be between 7 and 40"; //Simple Group
String formatErr="email:must be a well-formed email address"; //DetailedOnly Group
//when - validating against all groups
Set<ConstraintViolation<ContactPointDTO>> violations = validator.validate(
    invalidContact,
    SimplePlusDefault.class, DetailedOnly.class);
//then - all groups will have their violations reported
then(errors(violations)).contains(nameErr, sizeErr, formatErr).hasSize(3); ① ② ③
```

① `@NotNull` `name` (`nameError`) is part of `Default` group

② `@Size` `email` (`sizeError`) is part of `SimplePlusDefault` group

③ `@Email` `email` (`formatError`) is part of `DetailedOnly` group

392.3. Short-Circuit Validation

If we instead want to layer validations such that cheap validations come first and more extensive or expensive validations occur only after earlier groups are successful, we can define a `@GroupSequence`.

- Groups earlier in the sequence are performed first.
- Groups later in the sequence are performed later—but only if all constraints in earlier groups pass. Validation will short-circuit at the individual group level when applying a sequence.

The following snippet shows an example of defining a `@GroupSequence` that lists the order of group validation.

Example @GroupSequence

```
@GroupSequence({ SimplePlusDefault.class, DetailedOnly.class }) ①
public interface DetailOrder {};
```

① defines an order-list of validation groups to apply

The following example shows how the validation stopped at the `SimplePlusDefault` group and did not advance to the `DetailedOnly` group.

@GroupSequence Use Example

```
//when - validating using a @GroupSequence
violations = validator.validate(invalidContact, DetailOrder.class);
//then - validation stops once a group produces a violation
then(errors(violations)).contains(nameErr, sizeErr).hasSize(2); ①
```

① validation was short-circuited at the group where the first set of errors detected

392.4. Override Default Group



The `@GroupSequence` annotation can be directly applied to a type to override the default group when validating instances of that class.

Chapter 393. Spring Integration

We saw earlier how we could programmatically validate constraints for Java methods. This capability was not intended for business code to call — but rather for calls to be intercepted by AOP and constraints applied by that intercepting code. We can annotate `@Component` classes or interfaces with constraints and have Spring perform that validation role for us.

The following snippet shows an example of an interface with a simple `aCall` method that accepts an `int` parameter that must be greater than 5. All the information on the method call should be familiar to you by now. Only the `@Validated` annotation is new. The `@Validated` annotation triggers Spring AOP to apply Bean Validation to calls made on the type (interface or class).

Component Interface

```
import org.springframework.validation.annotation.Validated;
import jakarta.validation.constraints.Min;

@Validated ②
public interface ValidatedComponent {
    void aCall(@Min(5) int mustBeGE5); ①
}
```

① interface defines constraint for parameter(s)

② `@Validated` triggers Spring to perform method validation on component calls

393.1. Validated Component

The following snippet shows a class implementation of the interface and further declared as a `@Component`. Therefore, it can be injected and method calls subject to container interpose using AOP interception.

Component Implementation

```
@Component ①
public class ValidatedComponentImpl implements ValidatedComponent {
    @Override
    public void aCall(int mustBeGE5) {
    }
}
```

① designates this class implementation to be used for injection

The component is injected into clients.

Component Injection

```
@Autowired
private ValidatedComponent component;
```

393.2. ConstraintViolationException

With the component injected, we can have parameters and results validated against constraints.

The following snippet shows an example component call where a call is made with an invalid parameter. Spring performs the method validation, throws a `jakarta.validation.ConstraintViolationException`, and prevents the call. The `Set<ConstraintViolation>` can be obtained from the exception. At that point, we return to familiar territory we covered with programmatic validation.

Injected Component Validation by Spring

```
import jakarta.validation.ConstraintViolationException;  
...  
//when  
ConstraintViolationException ex = catchThrowableOfType(  
    () -> component.aCall(1), ①  
    ConstraintViolationException.class);  
//then  
Set<ConstraintViolation<?>> violations = ex.getConstraintViolations();  
String errorMsgs = violations.stream()  
    .map(v->v.getPropertyPath().toString() + ":" + v.getMessage())  
    .collect(Collectors.joining("\n"));  
then(errorMsgs).isEqualTo("aCall.mustBeGE5:must be greater than or equal to 5");
```

① Spring intercepts the component call, detects violations, and reports using exception

393.3. Successful Validation

Of course, if we pass valid parameter(s) to the method —

- the parameters are validated against the method constraints
- no exception is thrown
- the `@Component` method is invoked
- the return object is validated against declared constraints (none in this example)

Example Successful Call

```
assertThatNoException().isThrownBy(  
    ()->component.aCall(10) ①  
);
```

① parameter value 10 satisfies the `@Min(5)` constraint — thus no exception

393.4. Liskov Substitution Principle

One thing you may have noticed with the selected example is that the interface contained constraints and not the class declaration. As a matter of fact, if we add any additional constraint

beyond what the interface defined—we will get a `ConstraintDeclarationException` thrown—preventing the call from completing. The Bean Validation Specification [describes it](#) as following the [Liskov Substitution Principle](#)—where anything that is a subtype of `T` can be inserted in place of `T`. Said more specific to Bean Validation—a subtype or implementation class method cannot add more restrictive constraints to call.

```
@Validated
public interface ValidatedComponent {
    void aCall(@Min(5) int mustBeGE5);
}

@Component
public class ValidatedComponentImpl implements ValidatedComponent {
    @Override
    public void aCall(@Positive int mustBeGE5) {} //invalid ①
}
```

- ① Bean Validation enforces that subtypes cannot be more constraining than their interface or parent type(s)

Liskov Violation Error Message Example

```
jakarta.validation.ConstraintDeclarationException: HV000151: A method overriding another method must not redefine the parameter constraint configuration, but method ValidatedComponentImpl#aCall(int) redefines the configuration of ValidatedComponent#aCall(int).
```

393.5. Disabling Parameter Constraint Override

For the Hibernate Validator, the constraint override rule can be turned off during factory configuration. You can find other Hibernate-specific features in the [Hibernate Validator Specifics section](#) of the on-line documentation.

The snippet below uses a generic property interface to disable parameter override constraint.

Generic Property Setting

```
return Validation.byDefaultProvider() ①
    .configure()
    .addProperty("hibernate.validator.allow_parameter_constraint_override",
                 Boolean.TRUE.toString()) ②
    .parameterNameProvider(new MyParameterNameProvider())
    .buildValidatorFactory()
    .getValidator();
```

- ① generic factory configuration interface used to initialize factory

- ② generic property interface used to set custom behavior of Hibernate Validator

The snippet below uses a Hibernate-specific configurer and custom method to disable parameter override constraint.

Hibernate Specific API Supported Property Setting

```
return Validation.byProvider(HibernateValidator.class) ①
    .configure()
        .allowOverridingMethodAlterParameterConstraint(true) ②
        .parameterNameProvider(new MyParameterNameProvider())
    .buildValidatorFactory()
    .getValidator();
```

① Hibernate-specific configuration interface used to initialize factory

② Hibernate-specific method used to set custom behavior of Hibernate Validator

393.6. Spring Validated Group(s)

We saw earlier how we could programmatically validate using explicit validation groups. Spring uses the `@Validated` annotation in a dual role to define that as well.

- `@Validated` on the interface/class triggers validation to occur
- `@Validated` on a parameter or method causes the validation to apply the identified group(s)
 - the `groups` attribute is used for this purpose

Declarative Validation Group Assignment for Method

```
//@Validated ①
public interface PocService {
    @NotNull
    @Validated(CreatePlusDefault.class) ②
    PersonPocDTO createPOC(
        @NotNull ③
        @Valid PersonPocDTO personDTO); ④
```

① `@Validated` at the class/interface/component level triggers validation to be performed

② `@Validated` at the method level used to apply specific validation groups (`CreatePlusDefault`)

③ `@NotNull` at the property level requires `personDTO` to be supplied

④ `@Valid` at the property level triggered `personDTO` to be validated

393.7. Spring Validated Group(s) Example

The following snippet shows an example of a class where the `id` property is required to be null when validating against the `Create` group.

PersonPocDTO id Property

```
public class PersonPocDTO {
```

```
@Null(groups = Create.class, message = "cannot be specified for create")
private String id; ①
```

① `id` must be null only when validating against `Create` group

The following snippet shows the constrained method being passed a parameter that is illegal for the `Create` constraint group. A `ConstraintViolationException` is thrown with violations.

Spring Group Validation Example

```
PersonPocDTO pocWithId = pocFactory.make(oneUpId); ③
assertThatThrownBy(() -> pocService.createPOC(pocWithId)) ①
    .isInstanceOf(ConstraintViolationException.class)
    .hasMessageContaining("createPOC.person.id: cannot be specified for create");
②
```

① `@Validated` on component triggered validation to occur

② `@Validated(CreatePlusDefault.class)` caused `Create` and `Default` rules to be validated

③ poc instance created with an `id` assigned — making it invalid

Chapter 394. Custom Validation

Earlier I listed several common, [built-in constraints](#) and available [library constraints](#). Hopefully, they provide most or all of what is necessary to meet our validation needs—but there is always going to be that need for custom validation.

The snippet below shows an example of a custom validation being applied to a `LocalDate`—that validates the value is of a certain age in years, with an optional timezone offset.

@MinAge Custom Constraint Example Usage

```
public class ValidatedClass {  
    @MinAge(age = 16, tzOffsetHours = -4)  
    private LocalDate dob;
```

394.1. Constraint Interface Definition

We can start with the interface definition for our custom constraint annotation.

Example Validation Constraint Interface

```
@Documented  
@Target({ ElementType.METHOD, FIELD, ANNOTATION_TYPE, PARAMETER, TYPE_USE })  
@Retention( RetentionPolicy.RUNTIME )  
@Repeatable(value= MinAge.List.class)  
@Constraint(validatedBy = {  
    MinAgeLocalDateValidator.class,  
    MinAgeDateValidator.class  
})  
public @interface MinAge {  
    String message() default "age below minimum({age}) age";  
    Class<?>[] groups() default {};  
    Class<? extends Payload>[] payload() default {};  
  
    int age() default 0;  
    int tzOffsetHours() default 0;  
  
    @Documented  
    @Retention(RUNTIME)  
    @Target({ METHOD, FIELD, ANNOTATION_TYPE, PARAMETER, TYPE_PARAMETER })  
    @interface List {  
        MinAge[] value();  
    }  
}
```

394.2. @Documented Annotation

The `@Documented` annotation instructs the Javadoc processing to include the Javadoc for this annotation within the Javadoc output for the classes that use it.

@Documented Annotation

```
/**  
 * Defines a minimum age based upon a LocalDate, the current  
 * LocalDate, and a specified timezone.  
 */
```

```
@Documented //include this in Javadoc for elements that it is defined
```

The following images show the impact made to Javadoc for a different `@PersonHasName` annotation example. Not only are the constraints shown for the class, but the documentation for the annotations is included in the produced Javadoc.

Class PersonPocDTO

```
java.lang.Object  
info.ejava.examples.db.validation.contacts.dto.PersonPocDTO

---

@PersonHasName  
public class PersonPocDTO  
extends Object
```

Figure 177. PersonPocDTO Javadoc

```
@Documented  
@Constraint(validatedBy=PersonHasNameValidator.class)  
@Target(TYPE)  
@Retention(RUNTIME)  
public @interface PersonHasName
```

A person is required to have either a first or last name. One or the other can be null but not both.

Figure 178. `@PersonHasName` Annotation Javadoc

394.3. @Target Annotation

The `@Target` annotation defines locations where the constraint is legally allowed to be applied. The following table lists examples of the different target types.

Table 27. Annotation `@Target ElementType`s

<i>ElementType.FIELD</i>	<i>ElementType.METHOD</i>
<pre>@MinAge LocalDate dob;</pre>	<pre>@MinAge LocalDate getDob(); ① @MinAge void add(LocalDate dob, LocalDate dateOfHire); ②</pre>
define validation on a Java attribute within a class	<p>① <code>@MinAge</code> being used as return value constraint here</p> <p>② <code>@MinAge</code> being used as cross-param constraint here</p>
<i>ElementType.PARAMETER</i>	<i>ElementType.TYPE_USE</i>
<pre>void method(@MinAge LocalDate dob){}</pre>	<pre>List<@MinAge LocalDate> dobs;</pre>
define validation on a parameter to a method	define validation within a parameterized type
<i>ElementType.TYPE</i>	<i>ElementType.CONSTRUCTOR</i>
<pre>@MinAge class Person { LocalDate dob; }</pre>	<pre>class Person { LocalDate dob; @MinAge Person() {} }</pre>
define validation on an interface or class that likely inspects the state of the type	define validation on the resulting instance after constructor completes
<i>ElementType.ANNOTATION_TYPE</i>	
<pre>public @interface MinAge {} @MinAge(age=18) public @interface AdultAge {...</pre>	
This type allows other annotations to be defined based on this annotation. The snippet shows an example of constraint <code>@AdultAge</code> to be implemented as <code>@MinAge(age=18)</code>	

394.4. @Retention

`@Retention` is used to determine the lifetime of the annotation.

Annotation @Retention

```
@Retention(
```

```

//SOURCE - annotation discarded by compiler
//CLASS - annotation available in a class file but not loaded at runtime - default
RetentionPolicy.RUNTIME //annotation available through reflection at runtime
)

```

Bean Validation should always use **RUNTIME**

394.5. @Repeatable

The **@Repeatable** annotation and declaration of an annotation wrapper class is required to supply annotations multiple times on the same target. This is normally used in conjunction with different validation groups. The **@Repeatable.value** specifies an @interface that contains a **value** method that returns an array of the annotation type.

The snippet below provides an example of the **@Repeatable** portions of **MinAge**.

Enabling @Repeatable

```

@Repeatable(value= MinAge.List.class)
public @interface MinAge {
...
    @Retention(RUNTIME)
    @Target({ METHOD, FIELD, ANNOTATION_TYPE, PARAMETER, TYPE_USE })
    @interface List {
        MinAge[] value();
    }
}

```

The following snippet shows the annotation being applied multiple times to the same property—but assigned different groups.

Example @Repeatable Use

```

@MinAge(age=18, groups = {VotingGroup.class})
@MinAge(age=65, groups = {RetiringGroup.class})
public LocalDate getConditionalDOB() {
    return dob;
}

```

Repeatable Syntax Use Simplified

The requirement for the wrapper class is based on the Java requirement to have only one annotation type per target. Prior to Java 8, we were also required to explicitly use the construct in the code. Now it is applied behind the scenes by the compiler.



Pre-Java 8 Use of Repeatable

```

@MinAge.List({

```

```

    @MinAge(age=18, groups = {VotingGroup.class})
    @MinAge(age=65, groups = {RetiringGroup.class})
)
public LocalDate getConditionalDOB() {

```

394.6. @Constraint

The `@Constraint` is used to identify the class(es) that will implement the constraint. The annotation is not used for constraints built upon other constraints (e.g., `@AdultAge` ⇒ `@MinAge`). The annotation can specify multiple classes — one for each unique type the constraint can be applied to.

The following snippet shows two validation classes: one for `java.util.Date` and the other for `java.time.LocalDate`.

`@Constraint`

```

@Constraint(validatedBy = {
    MinAgeLocalDateValidator.class, ①
    MinAgeDateValidator.class ②
})
public @interface MinAge {

```

① validates annotated `LocalDate` values

② validates annotated `Date` values

Constraining Different Types

```

@MinAge(age=18, groups = {VotingGroup.class})
@MinAge(age=65, groups = {RetiringGroup.class})
public LocalDate getConditionalDOB() { ①
    return dob;
}

@MinAge(age=16, message="found java.util.Date age({age}) violation")
public Date getDobAsDate() { ②
    return Date.from(dob.atStartOfDay().toInstant(ZoneOffset.UTC));
}

```

① constraining type `LocalDate`

② constraining type `Date`

394.6.1. Core Constraint Annotation Properties

The core constraint annotation properties include

`message`

contains the default error message template to be returned when constraint violated. The contents of the message get `interpolated` to fill in variables and substitute entire text strings. This

provides a means for more detailed messages as well as internationalization of messages.

groups

identifies which group(s) to validate this constraint against

payload

used to supply instance-specific metadata to the validator. A common example is to establish a severity structure to instruct the validator how to react.

The following snippet provides an example declaration of core properties for `@MinAge` constraint.

Core Constraint Annotation Properties

```
public @interface MinAge {  
    String message() default "age below minimum({age}) age";  
    Class<?>[] groups() default {};  
    Class<? extends Payload>[] payload() default {};  
    ...
```

394.7. @MinAge-specific Properties

Each constraint annotation can also define its own unique properties. These values will be expressed in the target code and made available to the constraining code at runtime.

The following example shows the `@MinAge` constraint with two additional properties

- age - defines how old the subject has to be in years to be valid
- tzOffsetHours - an example property demonstrating we can have as many as we need

@MinAge-specific Properties

```
public @interface MinAge {  
    ...  
    int age() default 0;  
    int tzOffsetHours() default 0;  
    ...
```

394.8. Constraint Implementation Class

The annotation referenced zero or more constraint implementation classes — differentiated by the Java type they can process.

@Constraint

```
@Constraint(validatedBy = {  
    MinAgeLocalDateValidator.class,  
    MinAgeDateValidator.class  
})
```

```
public @interface MinAge {
```

Each implementation class has two methods they can override.

- `initialize()` accepts the specific annotation instance that will be validated against
- `isValid()` accepts the value to be validated and a context for this specific call. The minimal job of this method is to return true or false. It can optionally provide additional or custom details using the context.

394.9. Constraint Implementation Type Examples

The following snippets show the `@MinAge` constraint being implemented against two different temporal types: `java.time.LocalDate` and `java.util.Date`. We, of course, could have used inheritance to simplify the implementation.

@MinAge java.time.LocalDate Constraint Implementation Class

```
public class MinAgeLocalDateValidator implements ConstraintValidator<MinAge, LocalDate> {
    ...
    @Override
    public void initialize(MinAge annotation) { ... }
    @Override
    public boolean isValid(LocalDate dob, ConstraintValidatorContext context) { ... }
```

@MinAge java.util.Date Constraint Implementation Class

```
public class MinAgeDateValidator implements ConstraintValidator<MinAge, Date> {
    ...
    @Override
    public void initialize(MinAge annotation) { ... }
    @Override
    public boolean isValid(Date dob, ConstraintValidatorContext context) { ... }
```

394.10. Constraint Initialization

The constraint `initialize` provides a chance to validate whether the constraint definition is valid on its own. An invalid constraint definition is reported using a `RuntimeException`. If an exception is thrown during either the `initialize()` or `isValid()` method, it will be wrapped in a `ValidationException` before being reported to the application.

Constraint Initialization

```
public class MinAgeLocalDateValidator implements ConstraintValidator<MinAge, LocalDate> {
    private int minAge;
    private ZoneOffset zoneOffset;
```

```

@Override
public void initialize(MinAge annotation) {
    if (annotation.age() < 0) {
        throw new IllegalArgumentException("age constraint cannot be negative");
    }
    this.minAge = annotation.age();

    if (annotation.tzOffsetHours() > 23 || annotation.tzOffsetHours() < -23) {
        throw new IllegalArgumentException("tzOffsetHours must be between -23 and +23");
    }
    zoneOffset = ZoneOffset.ofHours(annotation.tzOffsetHours());
}

```

394.11. Constraint Validation

The `isValid()` method is required to return a boolean true or false—to indicate whether the value is valid, according to the constraint. It is a best-practice to only validate non-null values and to independently use `@NotNull` to enforce a required value.

The following snippet shows a simple evaluation of whether the expressed `LocalDate` value is older than the minimum required `age`.

Constraint Validation

```

@Override
public boolean isValid(LocalDate dob, ConstraintValidatorContext context) {
    if (null==dob) { //assume null is valid and use @NotNull if it should not be
        return true;
    }
    final LocalDate now = LocalDate.now(zoneOffset);
    final int currentAge = Period.between(dob, now).getYears();
    return currentAge >= minAge;
}

```



Treat Null Values as Valid

Null values should be considered valid and independently constrained by `@NotNull`.

394.12. Custom Violation Messages

I won't go into any detail here, but will point out that the `isValid()` method has the opportunity to either augment or replace the constraint violation messages reported.

The following example is from a cross-parameter constraint and is reporting that parameters 1 and 2 are not valid when used together in a method call.

Custom Violation Messages

```
context.buildConstraintViolationWithTemplate(context.getDefaultConstraintMessageTempla  
te())  
    .addParameterNode(1)  
    .addConstraintViolation()  
    .buildConstraintViolationWithTemplate(context  
        .getDefaultConstraintMessageTemplate())  
    .addParameterNode(2)  
    .addConstraintViolation();  
//the following removes default-generated message  
//context.disableDefaultConstraintViolation(); ①
```

- ① make this call to eliminate the default message

The following shows the default constraint message provided in the target code.

Default Constraint Message Provided

```
@ConsistentNameParameters(message = "name1 and/or name2 must be supplied") ①  
public NamedThing(String id, String name1, String name2, LocalDate dob) {
```

- ① `@ConsistentNameParameters` is a cross-parameter validation constraint validating name1 and name2

Generated Violation Message Paths

```
NamedThing.name1:name1 and/or name2 must be supplied ①  
NamedThing.name2:name1 and/or name2 must be supplied ①  
NamedThing.<cross-parameter>:name1 and/or name2 must be supplied ②
```

- ① path/message generated by the custom constraint validator
② default path/message generated by validation framework

Chapter 395. Cross-Parameter Validation

Custom validation is useful, but often times the customization is necessary for when we need to validate two or more parameters used together.

The following snippet shows an example of two parameters—name1 and name2—with the requirement that at least one be supplied. One or the other can be null—but not both.

Cross-Parameter Constraint Use Example

```
class NamedThing {  
    @ConsistentNameParameters(message = "name1 and/or name2 must be supplied") ①  
    public NamedThing(String id, String name1, String name2, LocalDate dob) {
```

① cross-parameter annotation placed on the method

395.1. Cross-Parameter Annotation

The cross-parameter constraint will likely only apply to a method or constructor, so the number of `@Targets` will be more limited. Other than that—the differences are not yet clear that it is performing special validation. Something else will be necessary in the `ConstraintValidator` class.

Cross-Parameter Annotation

```
@Documented  
@Constraint(validatedBy = ConsistentNameParameters.ConsistentNameParametersValidator  
.class)  
@Target({ElementType.METHOD, ElementType.CONSTRUCTOR})  
@Retention(RetentionPolicy.RUNTIME)  
public @interface ConsistentNameParameters {
```

395.2. @SupportedValidationTarget

Because of the ambiguity when annotating a method, we need to apply the `@SupportedValidationTarget` annotation to identify whether the validation is for the parameters going into the method or the response from the method.

- `ValidationTarget.PARAMETERS` - parameters to method
- `ValidationTarget.ANNOTATED_ELEMENT` - returned element from method

Example Cross-Parameter Validator Declaration

```
@SupportedValidationTarget(ValidationTarget.PARAMETERS) ①  
public class ConsistentNameParametersValidator  
    implements ConstraintValidator<ConsistentNameParameters, Object[]> { ②
```

① declaring that we are validating parameters going into method/ctor

- ② must accept `Object[]` that will be populated with actual parameters

@SupportValidationTarget adds Clarity to Annotation Purpose



Think how the framework would be confused without the `@SupportedValidationTarget` annotation if we wanted to validate a method that returned an `Object[]`. The framework would not know whether to pass us the parameters or the response object.

395.3. Method Call Correctness Validation

Funny - within the work of a validation method, it sometimes needs to validate whether it is being called correctly. Was the constraint annotation applied to a method with the wrong signature? Did — somehow — a parameter of the wrong type end up in an unexpected position?

The snippet below highlights the point that cross-parameter constraint validators are strongly tied to method signatures. They expect the parameters to be validated in a specific position in the array and to be of a specific type.

Validating Correct Method Call

```
@Override  
public boolean isValid(Object[] values, ConstraintValidatorContext context) { ①  
    if (values.length != 4) { ②  
        throw new IllegalArgumentException(  
            String.format("Unexpected method signature, 4 params expected, %d  
supplied", values.length));  
    }  
    for (int i=1; i<3; i++) { //look at positions 1 and 2 ③  
        if (values[i]!=null && !(values[i] instanceof String)) {  
            throw new IllegalArgumentException(  
                String.format("Illegal method signature, param[%d], String expected,  
%s supplied", i, values[i].getClass()));  
        }  
    }  
    ...
```

① method parameters supplied in `Object[]`

② not a specific requirement for this validation — but sanity check we have what is expected

③ names validated must be of type String

395.4. Constraint Validation

Once we have the constraint properly declared and call-correctness validated, the implementation will look similar to most other constraint validations. This method is required to return a true or false.

Constraint Validation

```
@Override  
public boolean isValid(Object[] values, ConstraintValidatorContext context) { ①  
    ...  
    String name1= (String) values[1];  
    String name2= (String) values[2];  
    return (StringUtils.isNotBlank(name1) || StringUtils.isNotBlank(name2));  
}
```

Chapter 396. Web API Integration

396.1. Vanilla Spring/AOP Validation

From what we have learned in the previous chapters, we know that we should be able to annotate any `@Component` class/interface—including a `@RestController`—and have constraints validated. I am going to refer to this as "Vanilla Spring/AOP Validation" because it is not unique to any component type.

The following snippet shows an example of the Web API `@RestController` that validates parameters according to `Create` and `Default` Groups.

@RestController Validating Constraints using Vanilla AOP Validation

```
@Validated ①
public class ContactsController {
    ...
    @RequestMapping(path=CONTACTS_PATH,
        method= RequestMethod.POST,
        consumes={MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE},
        produces={MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE})
    @Validated(PocValidationGroups.CreatePlusDefault.class) ②
    public ResponseEntity<PersonPocDTO> createPOC(
        @RequestBody
        @Valid ③
        PersonPocDTO personDTO) {
    ...
}
```

① triggers validation for component

② configures validator for method constraints

③ identifies constraint for parameter

If we call this with an invalid `personDTO` (relative to the Default or Create groups), we would expect to see validation fail and some sort of error response from the Web API.

396.2. ConstraintViolationException Not Handled

As expected—Spring will validate the constraints and throw a `ConstraintViolationException`. However, Spring Boot—out of the box—does not provide built-in exception advice for `ConstraintViolationException`. That will result in the caller receiving a `500/INTERNAL_SERVER_ERROR` status response with the default error reporting message. It is understandable that it would be the default since constraints can be technically validated and reported from all different levels of our application. The exception could realistically be caused by a real internal server error. However—the reported status does not always have to be generic and misleading.

```
> POST http://localhost:64153/api/contacts

{"id": "1", "firstName": "Douglass", "lastName": "Effertz", "dob": [2011, 6, 14], "contactPoints": [{"id": null, "name": "Cell", "email": "penni.kautzer@hotmail.com", "phone": "(876) 285-7887 x1055", "address": {"street": "69166 Angelo Landing", "city": "Jaredshire", "state": "IA", "zip": "81764-6850"}]}

< 500 INTERNAL_SERVER_ERROR Internal Server Error ①

{ "url" : "http://localhost:53298/api/contacts",
  "statusCode" : 500,
  "statusName" : "INTERNAL_SERVER_ERROR",
  "message" : "Unexpected Error",
  "description" : "unexpected error executing request:
jakarta.validation.ConstraintViolationException: createPOC.person.id: cannot be
specified for create",
  "timestamp" : "2021-07-01T14:58:48.777269Z" }
```

① INTERNAL_SERVER_ERROR status is misleading—cause is bad value provided by client

The violation—at least in this case—was a bad value from the client. The `id` property cannot be assigned when attempting to create a contact. Ideally—this would get reported as either a `400/BAD_REQUEST` or `422/UNPROCESSABLE_ENTITY`. Both are 4xx/Client error status and will point to something the client needs to correct.

396.3. ConstraintViolationException Exception Advice

Assuming that the invalid value came from the client, we can map the unhandled `ConstraintViolationException` to a `400/BAD_REQUEST` using (in this case) a global `@RestControllerAdvice`.

The following snippet shows how we can take some of the code we have seen used in the JUnit tests to report validation details—and use that within an `@ExceptionHandler` to extract the details and report as a `400/BAD_REQUEST` to the client.

Mapping `ConstraintViolationException` to `BAD_REQUEST`

```
import info.ejava.examples.common.web.BaseExceptionAdvice;
...
@RestControllerAdvice ①
public class ExceptionAdvice extends BaseExceptionAdvice { ②

    @ExceptionHandler(ConstraintViolationException.class)
    public ResponseEntity<MessageDTO> handle(ConstraintViolationException ex) {
        String description = ex.getConstraintViolations().stream()
            .map(v->v.getPropertyPath().toString() + ":" + v.getMessage())
            .collect(Collectors.joining("\n"));
        HttpStatus status = HttpStatus.BAD_REQUEST; ③
    }
}
```

```
        return buildResponse(status, "Validation Error", description, (Instant)null);
    }
```

- ① controller advice being applied globally to all controllers in the application context
- ② extending a class of exception handlers and helper methods
- ③ hard-wiring the exception to a **400/BAD_REQUEST** status

396.4. ConstraintViolationException Mapping Result

The following snippet shows the Web API response to the client expressed as a **400/BAD_REQUEST**.

ConstraintViolationException Mapped to 400/BAD_REQUEST

```
{ "url" : "http://localhost:53408/api/contacts",
  "statusCode" : 400,
  "statusName" : "BAD_REQUEST",
  "message" : "Validation Error",
  "description" : "createPOC.person.id: cannot be specified for create",
  "timestamp" : "2021-07-01T15:10:59.037162Z" }
```

Converting from a **500/INTERNAL_SERVER_ERROR** to a **400/BAD_REQUEST** is the minimum of what we wanted (at least it is a Client Error status), but we can try to do better. We understood what was requested—but could not process the payload as provided.

396.5. Controller Constraint Validation

To cause the violation to be mapped to a **422/UNPROCESSABLE_ENTITY** to better indicate the problem, we can activate validation within the controller framework itself versus the vanilla Spring/AOP validation.

The following snippet shows an example of the **@RestController** identifying validation and specific validation groups as part of the Web API framework. The **@Validated** annotation is now being used on the Web API parameters.

Activating @RestController Validation of Payload

```
@RequestMapping(path=CONTACTS_PATH,
    method= RequestMethod.POST,
    consumes={MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE},
    produces={MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE})
//@Validated(PocValidationGroups.CreatePlusDefault.class) -- no longer needed ①
public ResponseEntity<PersonPocDTO> createPOC(
    @RequestBody
    //@Valid -- replaced by @Validated ①
    @Validated(PocValidationGroups.CreatePlusDefault.class) ②
    PersonPocDTO personDTO) {
```

- ① vanilla Spring/AOP validation has been disabled
- ② Web API-specific parameter validation has been enabled

396.6. MethodArgumentNotValidException

Spring MVC will independently validate the `@RequestBody`, `@RequestParam`, and `@PathVariable` constraints according to internal rules. Spring will throw an `org.springframework.web.bind.MethodArgumentNotValidException` exception when encountering a violation with the request body. That exception is mapped—by default—to return a very terse `400/BAD_REQUEST` response.

The snippet below shows an example response payload for the default `MethodArgumentNotValidException` mapping.

MethodArgumentNotValidException Default Mapping

```
< 400 BAD_REQUEST Bad Request
{"timestamp":"2021-07-01T15:24:44.464+00:00",
 "status":400,
 "error":"Bad Request",
 "message":"",
 "path":"/api/contacts"}
```

By default—we may want to be terse to avoid too much information leakage. However, in this case, let's improve this.

396.7. MethodArgumentNotValidException Custom Mapping

Of course, we can change the behavior if desired using a custom exception handler.

The following snippet shows an example custom exception handler mapping `MethodArgumentNotValidException` to a `422/UNPROCESSABLE_ENTITY`.

MethodArgumentNotValidException Custom Mapping to 422/UNPROCESSABLE_ENTITY

```
@RestControllerAdvice
public class ExceptionAdvice extends BaseExceptionAdvice {
    @ExceptionHandler(ConstraintViolationException.class)
    public ResponseEntity<MessageDTO> handle(ConstraintViolationException ex) { ... }

    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<MessageDTO> handle(MethodArgumentNotValidException ex) { ①

        Stream<String> fieldMsgs = ex.getFieldErrors().stream() ②
            .map(e -> e.getObjectName() + ". " + e.getField() + ": " + e.getDefaultMessage());
        Stream<String> globalMsgs = ex.getGlobalErrors().stream() ③
            .map(e -> e.getObjectName() + ": " + e.getDefaultMessage());
```

```

        String description = Stream.concat(fieldMsgs, globalMsgs)
            .collect(Collectors.joining("\n"));
        return buildResponse(HttpStatus.UNPROCESSABLE_ENTITY, "Validation Error",
            description, (Instant)null);
    }

```

- ① Spring MVC throws `MethodArgumentNotValidException` for `@RequestBody` violations
- ② reports fields of objects in error
- ③ reports overall objects (e.g., cross-parameter violations) in error

396.7.1. MethodArgumentNotValidException Custom Mapping Response

This results in the client receiving an HTTP status indicating the request was understood, but the payload provided was invalid. The description is as terse or verbose as we want it to be.

MethodArgumentNotValidException Custom Mapping Response

```
{
    "url" : "http://localhost:53818/api/contacts",
    "statusCode" : 422,
    "statusName" : "UNPROCESSABLE_ENTITY",
    "message" : "Validation Error",
    "description" : "personPocDTO.id: cannot be specified for create",
    "timestamp" : "2021-07-01T15:38:48.045038Z"
}
```

Can Also Supply Client Value if Permitted



The exception handler has access to the invalid value if security policy allows information like that to be in the response. Note that error messages tend to be placed into logs and logs can end up getting handled at a generic level. For example, you would not want an invalid partial but mostly correct SSN to be part of an error log.

396.8. @PathVariable Validation

Note that the Web API maps the `@RequestBody` constraint violations independently of the other parameter types.

The following snippet shows an example of validation constraints applied to `@PathVariable`. These are physically in the URI.

@PathVariable Validation

```

@RequestMapping(path= CONTACT_PATH,
    method=RequestMethod.GET,
    produces={MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE})
public ResponseEntity<PersonPocDTO> getPOC(
    @PathVariable(name="id")
    @Pattern(regexp = "[0-9]+", message = "must be a number") ①
}

```

```
String id) {
```

① validation here is thru vanilla Spring/AOP validation

396.9. @PathVariable Validation Result

The Web API (using vanilla Spring/AOP validation here) throws a `ConstraintViolationException` for `@PathVariable` and `@RequestParam` properties. We can leverage the custom exception handler we already have in place to do a decent job reporting status.

The following snippet shows an example response that is being mapped to a `400/BAD_REQUEST` using our custom exception handler for `ConstraintViolationException`. `400/BAD_REQUEST` seems appropriate because the `id` path parameter is invalid garbage (`1...34`) in this case.

@PathVariable Validation Violation Response

```
> HTTP GET http://localhost:53918/api/contacts/1...34, headers={masked}
< BAD_REQUEST/400
{ "url" : "http://localhost:53918/api/contacts/1...34",
  "statusCode" : 400,
  "statusName" : "BAD_REQUEST",
  "message" : "Validation Error",
  "description" : "getPOC.id: must be a number",
  "timestamp" : "2021-07-01T15:51:34.724036Z" }
```

Remember—if we did not have that custom exception handler in place for `ConstraintViolationException`, the HTTP status would have been a `500/INTERNAL_SERVER_ERROR`.

Unmapped ConstraintViolationException for @PathVariable Violation

```
< 500 INTERNAL_SERVER_ERROR Internal Server Error
{"timestamp":"2021-07-01T19:21:31.345+00:00", "status":500, "error":"Internal Server Error", "message": "", "path":"/api/contacts/1...34"}
```

396.10. @RequestParam Validation

`@RequestParam` validation follows the same pattern as `@PathVariable` and gets reported using a `ConstraintViolationException`.

@RequestParam Validation

```
@RequestMapping(path= EXAMPLE_CONTACTS_PATH,
    method=RequestMethod.POST,
    consumes={MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE},
    produces={MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE})
public ResponseEntity<PersonsPageDTO> findPocsByExample(
    @RequestParam(value = "pageNumber", defaultValue = "0", required = false)
    @PositiveOrZero
```

```

    Integer pageNumber,
    @RequestParam(value = "pageSize", required = false)
    @Positive
    Integer pageSize,
    @RequestParam(value = "sort", required = false) String sortString,
    @RequestBody PersonPocDTO probe) {

```

396.11. @RequestParam Validation Violation Response

The following snippet shows an example response for an invalid set of query parameters.

@RequestParam Validation Violation Response

```

> POST http://localhost:53996/api/contacts/example?pageNumber=-1&pageSize=0
{ ... }
> BAD_REQUEST/400
{
  "url" : "http://localhost:53996/api/contacts/example?pageNumber=-1&pageSize=0", ① ②
  "statusCode" : 400,
  "statusName" : "BAD_REQUEST",
  "message" : "Validation Error",
  "description" : "findPocsByExample.pageNumber: must be greater than or equal to 0
\\nfindPocsByExample.pageSize: must be greater than 0",
  "timestamp" : "2021-07-01T15:55:44.089734Z" }

```

① pageNumber has an invalid negative value

② pageSize has an invalid non-positive value

396.12. Non-Client Errors

One thing you may notice with the previous examples is that every constraint violation was blamed on the client — whether it was bad server code calling internally or not.

As an example, let's have the API require that `value` be non-negative. A successful validation of that constraint will result in a service method call.

Web API Requires Client Supply Non-Negative @RequestParam

```

@RequestMapping(path = POSITIVE_OR_ZERO_PATH,
method=RequestMethod.GET,
produces = {MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE})
public ResponseEntity<?> positive(
    @PositiveOrZero ①
    @RequestParam(name = "value") int value) {
    PersonPocDTO resultDTO = contactsService.positiveOrZero(value); ②
}

```

- ① @RequestParam validated
- ② value from valid request passed to service method

396.13. Service Method Error

The following snippet shows that the service call makes an obvious error by passing the `value` to an internal component requiring the value to not be positive.

Downstream Component Makes Error

```
public class PocServiceImpl implements PocService {
    ...
    public PersonPocDTO positiveOrZero(int value) {
        //obviously an error!!
        internalComponent.negativeOrZero(value);
    ...
}
```

The internal component leverages the Bean Validation by placing a `@NegativeOrZero` constraint on the `value`. This is obviously going to fail when the value is ever non-zero.

Internal Component Declares Violated Constraint

```
@Component
@Validated
public class InternalComponent {
    public void negativeOrZero(@NegativeOrZero int value) {
```

396.14. Violation Incorrectly Reported as Client Error

The snippet below shows an example response of the internal error. It is being blamed on the client—when it was actually an internal server error.

Internal Server Error Incorrectly Reported as Client Error

```
> GET http://localhost:54298/api/contacts/positiveOrZero?value=1
< 400 BAD_REQUEST Bad Request
{ "url": "http://localhost:54298/api/contacts/positiveOrZero?value=1",
  "statusCode": 400,
  "statusName": "BAD_REQUEST",
  "message": "Validation Error",
  "description": "negativeOrZero.value: must be less than or equal to 0",
  "timestamp": "2021-07-01T16:23:27.666154Z"}
```

396.15. Checking Violation Source

One thing we can do to determine the proper HTTP response status—is to inspect the source

information of the violation.

The following snippet shows an example of inspecting whether the violation was reported by a class annotated with `@RestController`. If from the API, then report the `400/BAD_REQUEST` as usual. If not, report it as a `500/INTERNAL_SERVER_ERROR`. If you remember—that was the original default behavior.

Internal Error Detected through Source Information

```
@ExceptionHandler(ConstraintViolationException.class)
public ResponseEntity<MessageDTO> handle(ConstraintViolationException ex) {
    String description = ...

    boolean isFromAPI = ex.getConstraintViolations().stream() ①
        .map(v -> v.getRootBean().getClass().getAnnotation(RestController.class))
        .filter(a->a!=null)
        .findFirst()
        .isPresent();

    HttpStatus status = isFromAPI ?
        HttpStatus.BAD_REQUEST : HttpStatus.INTERNAL_SERVER_ERROR;
    return buildResponse(status, "Validation Error", description, (Instant)null);
}
```

① `isFromAPI` set to true if any of the violations came from a component annotated with `@RestController`

396.16. Internal Server Error Correctly Reported

The following snippet shows the response to the client when our exception handler detects that is handling at least one violation generated from a class annotated with `@RestController`.

Internal Server Error Correctly Reported

```
{ "url" : "http://localhost:54434/api/contacts/positiveOrZero?value=1",
  "statusCode" : 500,
  "statusName" : "INTERNAL_SERVER_ERROR",
  "message" : "Validation Error",
  "description" : "negativeOrZero.value: must be less than or equal to 0",
  "timestamp" : "2021-07-01T16:45:50.235724Z" }
```

Any Source of Constraint Violation May be used to Impact Behavior



There is no magic to using the `@RestController` annotation as a trigger for certain behavior. Annotations are used all the time to denote classes of a certain pattern. One could create a custom annotation that explicitly indicates what we are looking to identify.



Limit Externalizing Internal Implementation Details

The example provided internal server error details that could be of no use to the client and could be exposing internal implementation details. Strongly consider placing the details in a server-side log and state the basics to the client.

396.17. Service-detected Client Errors

Assuming we do a thorough job validating all client inputs at the `@RestController` level, we might be done. However, what about the case where the client validation is pushed down to the `@Service` components. We would have to adjust our violation source inspection.

The following snippet shows an example of a service validating client requests using the same constraints as before — except this is in a lower-level component.

Service Validating Client Request

```
public interface PocService {  
    @NotNull  
    @Validated(PocValidationGroups.CreatePlusDefault.class)  
    PersonPocDTO createPOC(  
        @NotNull  
        @Valid PersonPocDTO personDTO);
```

Without any changes, we get violations reported as `400/BAD_REQUEST` status — which, as I stated in the beginning, was "OK".

Service Violation Reported as 400/BAD_REQUEST

```
< 400 BAD_REQUEST Bad Request  
{ "url" : "http://localhost:55168/api/contacts",  
  "statusCode" : 400,  
  "statusName" : "BAD_REQUEST",  
  "message" : "Validation Error",  
  "description" : "createPOC.person.id: cannot be specified for create",  
  "timestamp" : "2021-07-01T17:40:12.221497Z" }
```

I won't try to improve the HTTP status using source annotations on the validating class. I have already shown how to do that. Let's try another technique.

396.18. Payload

One other option we have to is leverage the payload metadata in each annotation. `Payload` classes are interfaces extending `jakarta.validation.Payload` that identify certain characteristics of the constraint.

Annotations Include Payload Metadata

```
public @interface Xxx {  
    String message() default "...";
```

```
Class<?>[] groups() default { };  
Class<? extends Payload>[] payload() default { }; ①
```

① Annotations can carry extra metadata in the payload property

The snippet below shows an example of a Payload subtype that expresses the violation should be reported as a [500/INTERNAL_SERVICE_ERROR](#).

Example HttpStatus Payload

```
public interface InternalError extends Payload {}
```

This payload information can be placed in constraints that are known to be validated by internal components.

Internal Component with Payload

```
@Component  
@Validated  
public class InternalComponent {  
    public void negativeOrZero(@NegativeOrZero(payload = InternalError.class) int  
value) {
```

396.19. Exception Handler Checking Payloads

The snippet below shows our generic, global advice factoring in whether the violation came from an annotation with an [InternalError](#) in the payload.

Exception Handler Checking Payloads

```
@ExceptionHandler(ConstraintViolationException.class)  
public ResponseEntity<MessageDTO> handle(ConstraintViolationException ex) {  
    String description = ...;  
    boolean isFromAPI = ...;  
  
    boolean isInternalError = isFromAPI ? false : ①  
        ex.getConstraintViolations().stream()  
            .map(v -> v.getConstraintDescriptor().getPayload())  
            .filter(p-> p.contains(InternalError.class))  
            .findFirst()  
            .isPresent();  
  
    HttpStatus status = isFromAPI || !isInternalError ?  
        HttpStatus.BAD_REQUEST : HttpStatus.INTERNAL_SERVER_ERROR;  
  
    return buildResponse(status, "Validation Error", description, (Instant)null);  
}
```

① `isInternalError` set to true if any violations contain the [InternalError](#) payload

396.20. Internal Violation Exception Handler Results

The following snippet shows an example of a constraint violation where none of the violations were assigned a payload with `InternalError`. The status is returned as `400/BAD_REQUEST`.

Violation From Annotation without Payload

```
> POST http://localhost:55288/api/contacts
< 400/BAD_REQUEST
  "url" : "http://localhost:55288/api/contacts",
  "statusCode" : 400,
  "statusName" : "BAD_REQUEST",
  "message" : "Validation Error",
  "description" : "createPOC.person.id: cannot be specified for create",
  "timestamp" : "2021-07-01T17:56:23.080884Z"
}
```

The following snippet shows an example of a constraint violation where at least one of the violations were assigned a payload with `InternalError`. The client may not be able to make heads-or-tails out of the error message, but at least they would know it is something on the server-side to be corrected.

Violation from Annotation with InternalError Payload

```
> GET http://localhost:57547/api/contacts/positiveOrZero?value=1
< INTERNAL_SERVER_ERROR/500
{ "url" : "http://localhost:57547/api/contacts/positiveOrZero?value=1",
  "statusCode" : 500,
  "statusName" : "INTERNAL_SERVER_ERROR",
  "message" : "Validation Error",
  "description" : "negativeOrZero.value: must be less than or equal to 0",
  "timestamp" : "2021-07-01T20:25:05.188126Z" }
```

Chapter 397. JPA Integration

Bean Validation is integrated into the JPA standard. This can be used to validate entities mostly when created, updated, or deleted. Although not part of the standard, it is also used by some providers to customize generated database schema with additional RDBMS constraints (e.g., `@NotNull`, `@Size`). By default, the JPA provider will implement validation of the `Default` group for all `@Entity` classes during creation or update.

The following is a list of JPA properties that can be used to impact the behavior. They all need to be prefixed with `spring.jpa.properties.` when using Spring Boot properties to set the value.

Table 28. JPA Validation Configuration Properties

<code>jakarta.persistence.validation.mode</code>	ability to control validation at a high level	<ul style="list-style-type: none">• auto - implement validation if provider available (default)• callback - validation is required and will fail if the provider is missing• none - disable validation entirely within JPA
<code>jakarta.persistence.validation.group.pre-persist</code>	identify group(s) validated before inserting new row	<ul style="list-style-type: none">• <code>jakarta.validation.groups.Default.class</code> (default)
<code>jakarta.persistence.validation.group.pre-update</code>	identify group(s) validated before updating existing row	<ul style="list-style-type: none">• <code>jakarta.validation.groups.Default.class</code> (default)
<code>jakarta.persistence.validation.group.pre-remove</code>	identify group(s) validated before removing existing row	<ul style="list-style-type: none">• (none) (default)

Chapter 398. Mongo Integration

A basic Bean Validation implementation is integrated into Spring Data Mongo. It leverages event-specific callbacks from `AbstractMongoEventListener`, which is integrated into the Spring `ApplicationListener` framework.

There are no configuration settings, and after you see the details—you will quickly realize that they mean to handle the most common case (validate the `Default` group on `save()`) and for us to implement the corner-cases.

The following snippet shows an example of activating the default MongoRepository validation.

Basic MongoRepository Validation Configuration

```
import org.springframework.data.mongodb.core.mapping.event.ValidatingMongoEventListener;
...
@Configuration
public class MyMongoConfiguration {
    @Bean
    public ValidatingMongoEventListener mongoValidator(Validator validator) {
        return new ValidatingMongoEventListener(validator);
    }
}
```

398.1. Validating Saves

To demonstrate validation within the data tier, let's assume that our document class has a constraint for the `dob` to be supplied.

Example Mongo Document Class with Constraint

```
@Document(collection = "pocs")
public class PersonPOC {
    ...
    @NotNull
    private LocalDate dob;
    ...
}
```

When we attempt to save a PersonPOC in the repository without a `dob`, the following example shows that the Java source object is validated, a violation is detected, and a `ConstraintViolationException` is thrown.

Example Validation on save()

```
//given
PersonPOC noDobPOC = mapper.map(pocDTOFactory.make().withDob(null));
```

```
//when
assertThatThrownBy(() -> contactsRepository.save(noDobPOC))
    .isInstanceOf(ConstraintViolationException.class)
    .hasMessageContaining("dob: must not be null");
```

There is nothing more to it than that until we look into the implementation of [ValidatingMongoEventListener](#).

398.2. ValidatingMongoEventListener

[ValidatingMongoEventListener](#) extends [AbstractMongoEventListener](#), has a [Validator](#) from injection, and overrides a single event callback called [onBeforeSave\(\)](#).

ValidatingMongoEventListener

```
package org.springframework.data.mongodb.core.mapping.event;
...
public class ValidatingMongoEventListener extends AbstractMongoEventListener<Object> {
    ...
    private final Validator validator;
    @Override
    public void onBeforeSave(BeforeSaveEvent<Object> event) {
        ...
    }
}
```

It takes little imagination to guess how the rest of this works. I have removed the debug code from the method and provided the remaining details here.

onBeforeSave Details

```
@Override
public void onBeforeSave(BeforeSaveEvent<Object> event) {
    Set violations = validator.validate(event.getSource());

    if (!violations.isEmpty()) {
        throw new ConstraintViolationException(violations);
    }
}
```

The [onBeforeSaveEvent](#) is called after the source Java object has been converted to a form that is ready for storage.

398.3. Other AbstractMongoEventListener Events

There are many reasons—beyond validation, (e.g., sub-document ID generation)—we can take advantage of the [AbstractMongoEventListener callbacks](#), so it will be good to provide an overview of them now.

- There are three core events: Save, Load, and Delete
- Several **possible** stages to each core event
 - before action performed (e.g., delete)
 - and before converting between a Java object and **Document** (e.g., save and load)
 - and after converting between a Java object and **Document** (e.g., save and load)
 - after action is complete (e.g., save)

The following table lists the specific events.

Table 29. MongoMappingEvents

onApplicationEvent(MongoMappingEvent<?> event)	general purpose event handler
onBeforeConvert(BeforeConvertEvent<E> event)	callback before a Java object converted to Document
onBeforeSave(BeforeSaveEvent<E> event)	callback after Java object converted to Document and before saved to DB
onAfterSave(AfterSaveEvent<E> event)	callback after Document saved
onAfterLoad(AfterLoadEvent<E> event)	callback after Document loaded from DB and before converted to a Java object
onAfterConvert(AfterConvertEvent<E> event)	callback after Document converted to Java object
onBeforeDelete(BeforeDeleteEvent<E> event)	callback before document deleted from DB
onAfterDelete(AfterDeleteEvent<E> event)	callback after document deleted from DB

398.4. MongoMappingEvent

The **MongoMappingEvent** itself has three main items.

- Collection Name — name of the target collection
- Source — the source or target Java object
- **Document** — the source or target **bson** data type stored to the database

Our validation would always be against the **source**, so we just need a callback that provides us with a read-only value to validate.

Chapter 399. Patterns / Anti-Patterns

Every piece of software has an interface with some sort of pre-conditions and post-conditions that have some sort of formal or informal constraints. Constraint validation—whether using custom code or Bean Validation framework—is a decision to be made when forming the layers of a software architecture. The following patterns and anti-patterns list a few concerns to address. The original outline and content provided below is based on [Tom Hombergs' Bean Validation Anti-Patterns article](#).

399.1. Data Tier Validation

The Data tier has long been the keeper of data constraints—especially with RDBMS schema.

- Should the constraint validations discussed be implemented at that tier?
- Can validation wait all the way to the point where it is being stored?
- Should service and other higher levels of code be working with data that has not been validated?

399.1.1. Data Tier Validation Safety Checks

Hombergs' suggestion was to use the data tier validation as a safety check, but not the only layer.^[1]

That, of course, makes a lot of sense since the data tier may not need to know what a valid e-mail looks like or (to go a bit further) what type of e-mail addresses we accept? However, the data tier will want to sanity check that required fields exist and may want to go as far as validating format if query implementations require the data to be in a specific form.

399.2. Use case-specific Validation

Re-use is commonly a goal in software development. However, as we saw with validation groups—some data types have use case-specific constraints.

The simple example is when `id` could not be provided during a create but was legal in all other situations.

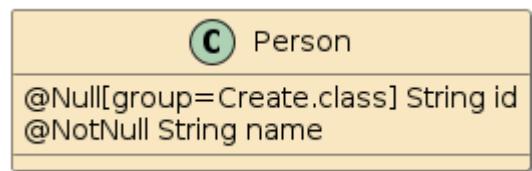


Figure 179. Re-usable Data Class with Use case-Specific Semantics

As more use case-specific constraints pile up on re-usable classes, they can get very cluttered and present a violation of single purpose [Single-responsibility principle](#).

399.2.1. Separate Syntactic from Semantic Validation

Hombergs proposes we

- use Bean Validation for syntactical validation for re-usable data classes
- implement query methods in the data classes for semantic state and perform checks against that specific state within the use case-specific code. ^[1]

One way of implementing use case-specific query methods and having them leverage Bean Validation constraints and a re-used data type would be to create use case-specific decorators or wrappers. [Lombok's experimental @Delegate](#) code generation may be of assistance here.

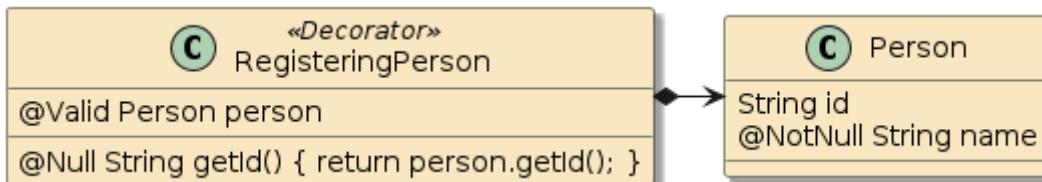


Figure 180. Use case-Specific Data Wrapper

399.3. Anti: Validation Everywhere

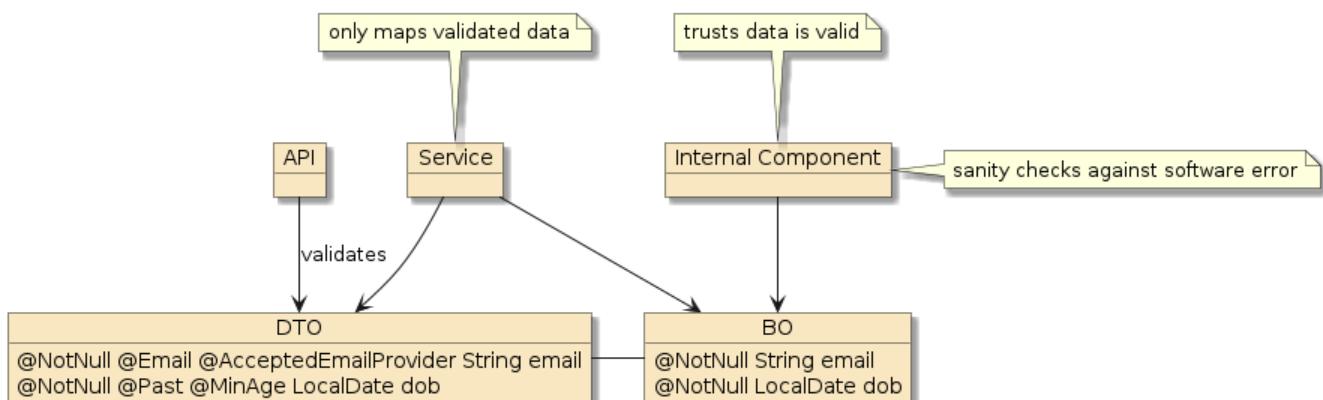
It is likely for us to want to validate at the client interface (Web API) since these are very external inputs. It is also likely for us to want to validate at the service level because our service could be injected into multiple client interfaces. It is then likely that internal components see how easy it is to add validation triggers and add to the mix. At the end of the line—the persistence layer adds a final check.

In some cases, we can get the same information validated several times. We have already shown in the Bean Validation details earlier in this topic—the challenge can be to determine what is a client versus internal issue when a violation occurs.

399.3.1. Establish Validation Architecture

Hombergs recommends having a clear validation strategy versus ad-hoc everywhere ^[1]

I agree with that strategy and like to have a clear dividing line of "once it reaches this point—data its valid". This is where I like to establish service entry points (validated) and internal components (sanity checked). Entry points check everything about the data. Internal components trust that the data given is valid and only need to verify if a programming error produced a null or some other common illegal value.



I also believe separating data types into external ("DTOs") and internal ("BOs") helps thin down the concerns. DTO classes would commonly be thorough and allow clients to know exactly what constraints exist. BO classes—used by the business and persistence logic only accept valid DTOs and should be done with detailed validation by the time they are mapped to BO classes.

399.3.2. Separating Persistence Concerns/Constraints

Hombergs went on to discuss a third tier of data types—persistence tier data types—separate from BOs as a way of separating persistence concerns away from BO data types.^[2] This is part of implementing a [Hexagonal Software Architecture](#) where the core application has no dependency on any implementation details of the other tiers. This is more of a plain software architecture topic than specific to validation—but it does highlight how there can be different contexts for the same conceptual type of data processed.

[1] ["Bean Validation Anti-Patterns"](#), Tom Hombergs

[2] ["Get Your Hands Dirty on Clean Architecture"](#), Tom Hombergs, 2019

Chapter 400. Summary

In this module, we learned:

- to add Bean Validation dependencies to the project
- to add declarative pre-conditions and post-conditions to components using the Bean Validation API
- to define declarative validation constraints
- to configure a `ValidatorFactory` and obtain a `Validator`
- to programmatically validate an object
- to programmatically validate parameters to and response from a method call
- to enable Spring/AOP validation for components
- to implement custom validation constraints
- to implement a cross-parameter validation constraint
- to configure Web API constraint violation responses
- to configure Web API parameter validation
- to configure JPA validation
- to configure Spring Data Mongo Validation
- to identify some patterns/anti-patterns for using validation

Unresolved directive in `jhu784-notes.adoc` - include::/builds/ejava-javaee/ejava-springboot-docs/courses/jhu784-notes/target/resources/docs/asciidoc/assignment6-async-{assignment6}.adoc[]

Porting to Spring Boot 3 / Spring 6

copyright © 2024 jim stafford (jim.stafford@jhu.edu)

Chapter 401. Introduction

This write-up documents key aspects encountered when porting the Ejava course examples from Spring Boot 2.7.x/Spring 5 to version 3.x/Spring 6.

401.1. Goals

The student will learn:

- specific changes required to port from Spring Boot 2.x to 3.x
- recognize certain error messages and map them to solutions

401.2. Objectives

At the conclusion of this lecture, the student will be able to:

1. update package dependencies

Chapter 402. Background

Spring Boot 3.0.0 (with Spring 6) was released in late Nov 2022. I initially ported the course examples from Spring Boot 2.7.0 (with Spring 5) to Spring Boot 3.0.2 (released Jan 20, 2023). Incremental releases of Spring Boot have been released in 2 to 4 week time periods since then. This writeup documents issues encountered—to include the initial signal of error and resolution taken.

Spring provides an official Migration Guide for [Spring Boot 3](#) and [Spring 6](#) that should be used as primary references.

The Spring Migration Guide [identifies](#) ways to enable some backward compatibility with Spring 5 or force upcoming compliance with Spring 6 with their [BeanInfoFactory](#) setting. I will not be discussing those options.

The change from Oracle (`javax.*`) to Jakarta (`jakarta.*`) enterprise APIs presents the simplest but most pervasive changes in the repository. Although the change is trivial and annoying—the enterprise `javax.*` APIs are frozen. All new enterprise API features will be added to the `jakarta.*` flavor of the libraries from here forward.

Refs:

- [spring-boot-dependencies:2.7.0](#)
- [spring-boot-dependencies:3.0.2](#)

Chapter 403. Preparation

There were two primary recommendations in the migration guide that were luckily addressed in the existing repository.

- migrate to Spring Boot 2.7.x
- use Java 17

Spring Boot 2.7.0 contained some deprecations that were also immediately addressed that significantly helped speed up the transition:

- Deprecation of `WebSecurityConfigurerAdapter` in favor of Component-based Web Security. `WebSecurityConfigurerAdapter` is now fully removed from Spring Boot 3/Spring 6.

Chapter 404. Dependency Changes

404.1. Spring Boot Version

The first and most obvious change was to change the `springboot.version` from `2.7.x` to `3.x`. This setting was in both `ejava-build-bom` (identifying `dependencyManagement`) and `ejava-build-parent` (identifying `pluginManagement`)

Spring Boot 2 Setting

```
<springboot.version>2.7.0</springboot.version>
```

Spring Boot 3 Setting

```
<springboot.version>3.0.2</springboot.version>
```

The version setting is used to import the targeted dependency definitions from `spring-boot-dependencies`.

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-dependencies</artifactId>
      <version>${springboot.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

404.2. JAXB DependencyManagement

The JAXB dependency definitions had to be spelled out in Spring 2.x like the following:

Spring Boot 2 JAXB dependencyManagement

```
<properties>
  <jaxb-api.version>2.3.1</jaxb-api.version>
  <jaxb-core.version>2.3.0.1</jaxb-core.version>
  <jaxb-impl.version>2.3.2</jaxb-impl.version>

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>javax.xml.bind</groupId>
      <artifactId>jaxb-api</artifactId>
      <version>${jaxb-api.version}</version>
    </dependency>
    <dependency>
      <groupId>com.sun.xml.bind</groupId>
      <artifactId>jaxb-core</artifactId>
```

```

<version>${jaxb-core.version}</version>
</dependency>
<dependency>
    <groupId>com.sun.xml.bind</groupId>
    <artifactId>jaxb-impl</artifactId>
    <version>${jaxb-impl.version}</version>
</dependency>

```

However, Spring Boot 3.x `spring-boot-dependencies` BOM includes a `jaxb-bom` that takes care of the JAXB dependencyManagement for us.

`ejava-build-bom.xml`

```

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-dependencies</artifactId> ①
            <version>${springboot.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>

```

① `ejava-build-bom` imports `spring-boot-dependencies`

`org.springframework.boot:spring-boot-dependencies-bom.xml`

```

<properties>
    <glassfish-jaxb.version>4.0.1</glassfish-jaxb.version>

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.glassfish.jaxb</groupId>
            <artifactId>jaxb-bom</artifactId> ①
            <version>${glassfish-jaxb.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>

```

① `spring-boot-dependencies` imports `jaxb-bom`

The `jaxb-bom` defines the replacement for the JAXB API using the `jakarta` naming. It also defines two versions of the `com.sun.xml.bind:jAXB-impl`. One uses the "old" `com.sun.xml.bind:jAXB-impl` naming construct and the other uses the "new" `org.glassfish.jaxb:jAXB-runtime` naming construct.

`org.glassfish.jaxb:jAXB-bom`

```

<dependencyManagement>
    <dependencies>

```

```

<dependency> <!--JAXB-API-->
  <groupId>jakarta.xml.bind</groupId>
  <artifactId>jakarta.xml.bind-api</artifactId> ①
  <version>${xml.bind-api.version}</version>
  <classifier>sources</classifier>
</dependency>

<!-- new -->
<dependency>
  <groupId>org.glassfish.jaxb</groupId>
  <artifactId>jaxb-runtime</artifactId> ②
  <version>${project.version}</version>
  <classifier>sources</classifier>
</dependency>

<!--OLD-->
<dependency>
  <groupId>com.sun.xml.bind</groupId>
  <artifactId>jaxb-impl</artifactId> ②
  <version>${project.version}</version>
  <classifier>sources</classifier>
</dependency>

```

① jaxb-bom defines artifact for JAXB API

② jaxb-bom defines old and new versions of artifact for JAXB runtime implementation

I am assuming the two ("old" and "new") are copies of the same artifact — and updated all runtime dependencies to the "new" **org.glassfish.jaxb** naming scheme.

404.3. Groovy

Class examples use a limited amount of groovy for Spock test framework examples. Version 3.0.x of **org.codehaus.groovy:groovy** was explicitly specified in **ejava-build-bom**.

Spring Boot 2.x ejava-build-bom Groovy Definition

```

<properties>
  <groovy.version>3.0.8</groovy.version>
<dependencyManagement>
  <dependencies>
    <dependency> ①
      <groupId>org.codehaus.groovy</groupId>
      <artifactId>groovy</artifactId>
      <version>${groovy.version}</version>
    </dependency>
  </dependencies>
</dependencyManagement>

```

① legacy **ejava-build-parent** explicitly defined groovy dependency

I noticed after the fact that Spring Boot 2.7.0 defined a version of **org.codehaus.groovy:groovy** that would have made the above unnecessary.

However, the move to Spring Boot 3 also caused a move in groups for groovy—from [org.codehaus.groovy](#) to [org.apache.groovy](#). The explicit dependency was removed from ejava-build-bom and dependencies updated to groupId [org.apache.groovy](#).

Spring Boot 3 spring-boot-dependencies.pom

```
<properties>
    <groovy.version>4.0.7</groovy.version>
<dependencyManagement>
    <dependencies>
        <dependency> ①
            <groupId>org.apache.groovy</groupId>
            <artifactId>groovy-bom</artifactId>
            <version>${groovy.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
```

① `spring-boot-dependencies` imports `groovy-bom`

groovy-bom

```
<dependencyManagement>
    <dependencies>
        <dependency> ①
            <groupId>org.apache.groovy</groupId>
            <artifactId>groovy</artifactId>
            <version>4.0.7</version>
        </dependency>
```

① `groovy-bom` now used to explicitly define groovy dependency

```
[ERROR]      'dependencies.dependency.version' for org.codehaus.groovy:groovy:jar is
missing. @ info.ejava.examples.build:ejava-build-parent:6.1.0-SNAPSHOT,
/Users/jim/proj/ejava-javae/ejava-springboot/build/ejava-build-parent/pom.xml, line
438, column 29
```

404.4. Spock

Using Spock test framework with Spring Boot 3.x requires the use of Spock version \geq 2.4-M1 and groovy 4.0. I also noted that the M1 version for 2.4 was required to work with `@SpringBootTest`.

Spring Boot 2 ejava-build-bom Spock property spec

```
<properties>
    <spock.version>2.0-groovy-
3.0</spock.version>
```

Spring Boot 3 ejava-build-bom Spock property spec

```
<properties>
    <spock.version>2.4-M1-groovy-
4.0</spock.version>
```

The above property definition is seamlessly used to define the necessary dependencies in the following snippet in order to use Spock test framework.

ejava-build-bom Spock Definition

```
<properties>
  <spock.version>...</spock.version>
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.spockframework</groupId>
      <artifactId>spock-bom</artifactId>
      <version>${spock.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
    <dependency>
      <groupId>org.spockframework</groupId>
      <artifactId>spock-spring</artifactId>
      <version>${spock.version}</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

404.5. Flapdoodle

Direct support for the Flapdoodle embedded Mongo database was removed from Spring Boot 3, but can be manually brought in with the following definition for "spring30x".

Spring Boot 3.x ejava-build-bom

```
<properties>
  <flapdoodle.spring30x.version>4.5.2</flapdoodle.spring30x.version>
<dependencyManagement>
  <dependencies>
    <dependency> ①
      <groupId>de.flapdoodle.embed</groupId>
      <artifactId>de.flapdoodle.embed.mongo.spring30x</artifactId>
      <version>${flapdoodle.spring30x.version}</version>
    </dependency>
  </dependencies>
</dependencyManagement>
```

① **spring-boot-dependencies** no longer defines the flapdoodle dependency. New Spring Boot 3.x flapdoodle dependency now defined by **ejava-build-bom**

Of course the dependency declaration groupId must be changed from **de.flapdoodle.embed.mongo** to **de.flapdoodle.embed.mongo.spring30x** in the child projects as well.

Spring Boot 2.x Flapdoodle Declaration

```
<dependency>
  <groupId>de.flapdoodle.embed</groupId>
```

```

<artifactId>de.flapdoodle.embed.mongo</artifactId>
<scope>test</scope>
</dependency>

```

Spring Boot 3.x Flapdoodle Spring 30x Declaration

```

<dependency>
  <groupId>de.flapdoodle.embed</groupId>
  <artifactId>de.flapdoodle.embed.mongo.spring30x</artifactId>
  <scope>test</scope>
</dependency>

```

404.6. HttpClient / SSL

The ability to define property features to outgoing HTTP connections requires use of the `org.apache.httpcomponents` libraries.

Spring Boot 2.x HttpComponents Dependency Definition

```

<properties>
  <httpClient.version>4.5.13</httpClient.version>
<dependencyManagement>
  <dependencies>
    <dependency> ①
      <groupId>org.apache.httpcomponents</groupId>
      <artifactId>httpclient</artifactId>
      <version>${httpClient.version}</version>
    </dependency>
  </dependencies>
</dependencyManagement>

```

① Spring Boot 2.x `spring-boot-dependencies` defined dependency on `httpclient`

Spring Boot 3 has migrated to "client5" version of the libraries. The older version gets replaced with the following. The dependencyManagement definition for httpclient(anything) can be removed from our local `ejava-build-bom`.

Spring Boot 3.x HttpComponents Client 5 Dependency Definition

```

<properties>
  <httpClient5.version>5.1.4</httpClient5.version>
<dependencyManagement>
  <dependencies>
    <dependency> ①
      <groupId>org.apache.httpcomponents.client5</groupId>
      <artifactId>httpclient5</artifactId>
      <version>${httpClient5.version}</version>
    </dependency>
  </dependencies>
</dependencyManagement>

```

① Spring Boot 3.x `spring-boot-dependencies` defines dependency on `httpclient5`

However, `httpclient5` requires a secondary library to configure SSL connections. `ejava-build-bom` now defines the dependency for that.

Spring Boot 3.x HttpComponents Client 5 Dependency - ejava-build-bom

```
<properties>
    <sslcontext-kickstart.version>7.4.9</sslcontext-kickstart.version>
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>io.github.hakky54</groupId> ①
            <artifactId>sslcontext-kickstart-for-apache5</artifactId>
            <version>${sslcontext-kickstart.version}</version>
        </dependency>
```

① `ejava-build-bom` defines necessary dependency to configure `httpclient5` SSL connections

404.7. Javax / Jakarta Artifact Dependencies

With using the spring-boot-starters, there are very few direct dependencies on enterprise artifacts. However, for the direct API references—simply change the `javax.*` groupId to `jakarta.*`.

Spring Boot 2.x API Dependency Definition

```
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-
api</artifactId>
    <scope>provided</scope>
</dependency>
```

Spring Boot 3.x API Dependency Definition

```
<dependency>
    <groupId>jakarta.servlet</groupId>
    <artifactId>jakarta.servlet-
api</artifactId>
    <scope>provided</scope>
</dependency>
```

404.8. jakarta.inject

`javax.inject` does not have a straight replacement and is not defined within the Spring Boot BOM. Replace any `javax.inject` dependencies with `jakarta.inject-api`.

javax.inject-api

```
<dependency>
    <groupId>javax.inject</groupId>
    <artifactId>javax.inject</artifactId>
    <version>1</version>
</dependency>
```

jakarta.inject-api

```
<dependency>
    <groupId>jakarta.inject</groupId>
```

```
<artifactId>jakarta.inject-api</artifactId>
<version>2.0.1</version>
</dependency>
```

Since `spring-boot-dependencies` does not provide a `dependencyManagement` entry for `inject`—it was difficult to determine which version would be best appropriate. I went with `2.0.1` and added to the `ejava-build-bom`.

`ejava-build-bom:jakarta.inject-api`

```
<properties>
  <jakarta.inject-api.version>2.0.1</jakarta.inject-api.version>
<build>
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>jakarta.inject</groupId>
        <artifactId>jakarta.inject-api</artifactId>
        <version>${jakarta.inject-api.version}</version>
      </dependency>
```

404.9. ActiveMQ / Artemis Dependency Changes

ActiveMQ does not yet support `jakarta` packaging, but its `artemis` sibling does. Modify all local pom dependency definitions to the `artemis` variant to first get things compiling. Dependency management will be taken care of by the Spring Boot Dependency POM.

`Spring Boot 2.x ActiveMQ Dependency - local child pom.xml`

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-activemq</artifactId>
</dependency>
```

`Spring Boot 3.x Artemis Dependency - local child pom.xml`

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-artemis</artifactId>
</dependency>
```

Chapter 405. Package Changes

Once maven artifact dependencies are addressed, resolvable Java package imports can be put in place. `javax` APIs added to the Java language (e.g., `javax.sql.*`) are still in the `javax` structure. `javax` APIs that are part of independent Enterprise APIs have been moved to `jakarta`.

The table below summarizes the APIs encountered in the EJava course examples repository. Whether through command-line (e.g., `find`, `sed`) or IDE search/replace commands—it is well worth the time to identify a global way to make these mindless `javax.(package)` to `jakarta.(package)` package name changes.

Spring Boot 2.x Enterprise Package Imports

```
import javax.annotation.*;
import javax.persistence.*;
import javax.jms.*;
import javax.servlet.*;
import javax.validation.*;
import javax.xml.bind.annotation.*;
```

Spring Boot 3.x Enterprise Package Imports

```
import jakarta.annotation.*;
import jakarta.jms.*;
import jakarta.persistence.*;
import jakarta.servlet.*;
import jakarta.validation.*;
import jakarta.xml.bind.annotation.*;
```

For example, the following bash script will locate all Java source files with `import javax` and change those occurrences to `import jakarta`.



Baseline changes prior to bulk file changes

Baseline all in-progress changes prior to making bulk file changes so that you can easily revert to the previous state.

change occurrences of "import javax" with "import jakarta"

```
$ for file in `find . -name "*.java" -exec grep -l 'import javax' {} \;`; do sed -i '' 's/import javax/import jakarta/' $file; done
```

However, not all `javax` packages are part of JavaEE. We now need to execute the following to correct `javax.sql`, `javax.json`, and `javax.net`, imports caught up in the mass change.

revert occurrences of "import jakarta" back to "import javax"

```
$ for file in `find . -name "*.java" -exec grep -l 'import jakarta.sql' {} \;`; do sed -i '' 's/import jakarta.sql/import javax.sql/' $file; done

$ for file in `find . -name "*.java" -exec grep -l 'import jakarta.json' {} \;`; do sed -i '' 's/import jakarta.json/import javax.json/' $file; done

$ for file in `find . -name "*.java" -exec grep -l 'import jakarta.net' {} \;`; do sed -i '' 's/import jakarta.net/import javax.net/' $file; done
```

Chapter 406. AssertJ Template Changes

AssertJ test assertion library has the ability to generate type-specific assertions. However, some of the generated classes make reference to deprecated `javax.*` packages ...

AssertJ assertion generator, build-in template

```
@javax.annotation.Generated(value="assertj-assertions-generator") ①
public class BddAssertions extends org.assertj.core.api.BDDAssertions {
    ...
}
```

① `javax` package name prefix must be renamed

... and must be updated to `jakarta.*`.

Required template modification

```
@jakarta.annotation.Generated(value="assertj-assertions-generator") ①
public class BddAssertions extends org.assertj.core.api.BDDAssertions {
    ...
}
```

① `jakarta` package name prefix must be used in place of `javax`

However, AssertJ assertions generator `releases` have been idle since Feb 2021 (version [2.2.1](#)) and our only option is to manually edit the templates ourself.

The testing with AssertJ assertions lecture notes covers how to customize the generator.

Review: AssertJ assertion generator template customizations

```
$ ls app/app-testing/apptesting-testbasics-example/src/test/resources/templates/
sort
ejava_bdd_assertions_entry_point_class_template.txt ①
```

① template was defined for custom type

Review: AssertJ assertion generator maven configuration

```
<!-- generate custom AssertJ assertions -->
<plugin>
    <groupId>org.assertj</groupId>
    <artifactId>assertj-assertions-generator-maven-plugin</artifactId>
    <configuration>
        <classes> ①
            <param>info.ejava.examples.app.testing.testbasics.Person</param>
        </classes>
        <templates>
            <!-- local customizations -->
            <templatesDirectory>
                ${basedir}/src/test/resources/templates/</templatesDirectory>
```

```
<bddEntryPointAssertionClass>ejava_bdd_assertions_entry_point_class_template.txt</bddE  
ntryPointAssertionClass>  
    </templates>  
  </configuration>  
</plugin>
```

① custom template and type was declared with **AssertJ** plugin

The following listing show we can host [downloaded](#) and modified template files.

Modified AssertJ generator plugin template files

```
$ ls app/app-testing/apptesting-testbasics-example/src/test/resources/templates/ |  
sort  
ejava_bdd_assertions_entry_point_class_template.txt  
jakarta_bdd_soft_assertions_entry_point_class_template.txt  
jakarta_custom_abstract_assertion_class_template.txt  
jakarta_custom_assertion_class_template.txt  
jakarta_custom_hierarchical_assertion_class_template.txt  
jakarta_junit_soft_assertions_entry_point_class_template.txt  
jakarta_soft_assertions_entry_point_class_template.txt  
jakarta_standard_assertions_entry_point_class_template.txt
```

The following snippet shows how we can configure the plugin to use the additional custom template files.

Declared modified AssertJ generator plugin template files

```
<!-- Spring Boot 3.x / AspectJ jakarta customizations -->  
<!-- https://github.com/assertj/assertj-assertions-generator-maven-plugin/issues/93  
-->  
<assertionClass>jakarta_custom_assertion_class_template.txt</assertionClass>  
<assertionsEntryPointClass>jakarta_standard_assertions_entry_point_class_template.txt<  
/assertionsEntryPointClass>  
<hierarchicalAssertionAbstractClass>jakarta_custom_abstract_assertion_class_template.t  
xt</hierarchicalAssertionAbstractClass>  
<hierarchicalAssertionConcreteClass>jakarta_custom_hierarchical_assertion_class_temp  
late.txt</hierarchicalAssertionConcreteClass>  
<softEntryPointAssertionClass>jakarta_soft_assertions_entry_point_class_template.txt</  
softEntryPointAssertionClass>  
<junitSoftEntryPointAssertionClass>jakarta_junit_soft_assertions_entry_point_class_t  
emplate.txt</junitSoftEntryPointAssertionClass>
```

Chapter 407. Spring Boot Configuration Property

@ConstructorBinding is used to designate how to populate the properties object with values. In Spring Boot 2.x, the annotation could be applied to the class or constructor.

Spring Boot 2 ConfigurationProperties Constructor Binding Option

```
import org.springframework.boot.context.properties.ConstructorBinding;

@ConstructorBinding ①
public class AddressProperties {
    private final String street;
    public AddressProperties(String street, String city, String state, String zip) {
    ...
}
```

① annotation could be applied to class or constructor

In Spring Boot 3.x, the annotation has been moved one Java package level lower—into the bind package—and the new definition can only be legally applied to constructors.

Spring Boot 3 ConfigurationProperties Constructor Binding Option

```
import org.springframework.boot.context.properties.bind.ConstructorBinding; ①

public class AddressProperties {
    private final String street;
    @ConstructorBinding ②
    public AddressProperties(String street, String city, String state, String zip) {
    ...
}
```

① annotation moved to new package

② annotation can only be applied to a specific constructor

However, it is technically only needed when there are multiple constructors.

@ConstructorBinding is only needed when having multiple constructors

```
@ConstructorBinding //only required for multiple constructors
public BoatProperties(String name) {
    this.name = name;
}
//not used for ConfigurationProperties initialization
public BoatProperties() { this.name = "default"; }
```

Chapter 408. HttpStatus

`HttpStatus` represents the status returned from an HTTP call. Responses are primarily in the 1xx, 2xx, 3xx, 4xx, and 5xx ranges with some well-known values.

When updating to Spring Boot 3, you may encounter the following compilation problem:

HttpStatus compilation error

```
incompatible types: org.springframework.http.HttpStatusCode cannot be converted to  
org.springframework.http.HttpStatus
```

408.1. Spring Boot 2.x

Spring Boot 2.x used an Enum type to represent these well-known values and properties and all interfaces accepted and returned that enum type.

Spring 5.x HttpStatus enum

```
public enum HttpStatus {  
    OK(200, Series.SUCCESSFUL, "OK")  
    CREATED(201, Series.SUCCESSFUL, "Created"),  
    ...
```

The following are two code examples for acquiring an `HttpStatus` object:

Getting HttpStatus enum from ClientHttpResponse

```
public class ResponseEntity<T> extends HttpEntity<T> {  
    public HttpStatus getStatusCode() {  
  
    - - -  
    ClientHttpResponse response = ...  
    HttpStatus status = response.getStatusCode();
```

Getting HttpStatus enum from ClientHttpResponse and HttpStatusCodeException

```
//when  
HttpStatus status;  
try {  
    status = homesClient.hasHome("anId").getStatusCode();  
} catch (HttpStatusCodeException ex) {  
    status = ex.getStatusCode();  
}
```

The following is an example of inspecting the legacy `HttpStatus` object.

Inspecting HttpStatus enum

```
then(response.getStatusCode().series()).isEqualTo(HttpStatus.Series.SUCCESSFUL);
```

The problem was that the HttpStatus enum could not represent custom HTTP status values.

408.2. Spring Boot 3.x

Spring 6 added a breaking change by having methods accept and return a new `HttpStatusCode` interface. The `HttpStatus` enum now implements that interface but cannot be directly resolved to an `HttpStatus` without an additional lookup.

Spring 6 HttpStatus Implements HttpStatusCode

```
public enum HttpStatus implements HttpStatusCode {
    OK(200, Series.SUCCESSFUL, "OK")
    CREATED(201, Series.SUCCESSFUL, "Created"),
    ...
}
```

One needs to call a lookup method to convert to an `HttpStatus` instance if the `(HttpStatusCode` object is needed. Use `resolve()` if null is acceptable (e.g., log statements) and `valueOf()` if required.

Getting HttpStatus type from ClientHttpResponse

```
public class ResponseEntity<T> extends HttpEntity<T> {
    public HttpStatusCode getStatusCode() {
        ...
        ClientHttpResponse response = ...
        HttpStatus status = HttpStatus.resolve(response.getStatusCode().value()); ①
        //or
        HttpStatus status = HttpStatus.valueOf(response.getStatusCode().value()); ②
    }
}
```

① returns null if value is not resolved

② throws IllegalArgumentException if value is not resolved

Getting HttpStatusCode enum from ClientHttpResponse and HttpStatusCodeException

```
HttpStatusCode status;
try {
    status = homesClient.hasHome("anId").getStatusCode();
} catch (HttpStatusCodeException ex) {
    status = ex.getStatusCode();
}
```

Accessing HttpStatus methods

```
then(response.getStatusCode().is2xxSuccessful()).isTrue();
```

Most of the same information is available — just not as easy to get to.

Chapter 409. HttpMethod

The common values for HttpMethod are also very well-known.

409.1. Spring Boot 2.x

Spring Boot 2.x used an enum to represent these well-known values and associated properties.

Spring Boot 2.x/Spring 5 HttpMethod as an Enum

```
public enum HttpMethod {  
    GET, HEAD, POST, PUT, PATCH, DELETE, OPTIONS, TRACE;  
    ...
```

Leveraging HttpMethod enum Definition for @ParameterizedTest value source

```
@ParameterizedTest  
 @EnumSource(value = HttpMethod.class, names = {"GET", "POST"})  
 void anonymous_user(HttpMethod method) {
```

The rigid aspects of the enum made it not usable for custom HTTP methods.

409.2. Spring Boot 3.x

Spring Boot 3.x changed HttpMethod from an enum to a regular class as well.

```
public final class HttpMethod implements Comparable<HttpMethod>, Serializable {  
    public static final HttpMethod GET = new HttpMethod("GET");  
    public static final HttpMethod POST = new HttpMethod("POST");  
    ...
```

The following shows the `@ParameterizedTest` from above, updated to account for the change. `@EnumSource` was changed to `@CsvSource` and the provided String was converted to an HttpMethod type within the method.

Leveraging HttpMethod String values and Conversion for @ParameterizedTest value source

```
@ParameterizedTest  
 @ValuesSource(strings={"GET", "POST"})  
 void anonymous_user(String methodName) {  
     HttpMethod method = HttpMethod.valueOf(methodName);
```

Chapter 410. Spring Factories Changes

The location for AutoConfiguration bootstrap classes has changed from the general-purpose **META-INF/spring.factories...**

META-INF/spring.factories

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\  
    info.ejava.examples.app.hello.HelloAutoConfiguration, \  
    info.ejava.examples.app.hello.HelloResourceAutoConfiguration
```

to the bootstrap-specific **META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports** file

META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports

```
info.ejava.examples.app.hello.HelloAutoConfiguration  
info.ejava.examples.app.hello.HelloResourceAutoConfiguration
```

The same information is conveyed in the **import** file — just expressed differently.

META-INF/spring.factories still exists. It is no longer used to express this information.

Chapter 411. Spring WebSecurityConfigurerAdapter

Spring had deprecated `WebSecurityConfigurerAdapter` by the time Spring Boot 2.7.0 was released.

Spring Boot 2 Deprecated WebSecurityConfigurerAdapter

```
@Configuration
@Order(100)
@ConditionalOnClass(WebSecurityConfigurerAdapter.class)
public static class SwaggerSecurity extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.requestMatchers(cfg->cfg
            .antMatchers("/swagger-ui*", "/swagger-ui/**", "/v3/api-docs/**"));
        http.csrf().disable();
        http.authorizeRequests(cfg->cfg.anyRequest().permitAll());
    }
}
```

The deprecated `WebSecurityConfigurerAdapter` approach was replaced with the Component-based `@Bean` factory mechanism. Spring 6 has completely eliminated the adapter approach.

Spring Boot Component-based

```
@Bean
@Order(100)
public SecurityFilterChain swaggerSecurityFilterChain(HttpSecurity http) throws
Exception {
    http.securityMatchers(cfg->cfg
        .requestMatchers("/swagger-ui*", "/swagger-ui/**", "/v3/api-docs/**"))
);
    http.csrf().disable();
    http.authorizeHttpRequests(cfg->cfg.anyRequest().permitAll());
    return http.build();
}
```

411.1. SecurityFilterChain securityMatcher

One or more request matchers can be applied to the `SecurityFilterChain`, primarily for the cases when there are multiple `SecurityFilterChains`. Lacking a request matcher—the highest priority `SecurityFilterChain` will likely process all requests. For Spring Boot 2.x/Spring 5, this was expressed with a `requestMatchers()` builder call on the injected `HttpSecurity` object.

```
@Bean
@Order(50)
public SecurityFilterChain h2Configuration(HttpSecurity http) throws Exception {
```

```
http.requestMatchers(cfg->...); ①
```

```
...
```

① `SecurityFilterChain.requestMatchers()` determined what filter chain will process

In Spring Boot 3/Spring 6, the request matcher for the `SecurityFilterChain` is now expressed with a `securityMatchers()` call. They function the same with a different name to help distinguish the call from the ones made to configure `RequestMatcher`.

```
@Bean  
@Order(50)  
public SecurityFilterChain h2Configuration(HttpSecurity http) throws Exception {  
    http.securityMatchers(cfg->...); ①  
    ...
```

① `securityMatchers()` replaces `requestMatchers()` for `SecurityFilterChain`

A simple search for `requestMatchers` and replace with `securityMatchers` is a suitable solution.

Search/Replace requestMatchers with securityMatchers Builder

```
for file in `find . -name "*.java" -exec grep -l 'requestMatchers(' {} \;`; do sed -i  
'` 's/requestMatchers(/securityMatchers(/' $file; done
```

411.2. antMatchers/requestMatchers

The details of the `RequestMatcher` for both the `SecurityFilterChain` and `WebSecurityCustomizer` were defined by a `antMatchers()` builder. The `mvcMatchers()` builder also existed, but were not used in the course examples.

```
@Bean  
@Order(50)  
public SecurityFilterChain h2Configuration(HttpSecurity http) throws Exception {  
    http.requestMatchers(cfg->cfg.antMatchers( ①  
        "/h2-console/**", "/login", "/logout"));  
    ...  
  
    @Bean  
    public WebSecurityCustomizer authzStaticResources() {  
        return (web) -> web.ignoring().antMatchers( ①  
            "/content/**");  
    }
```

① legacy `antMatchers()` defined details of legacy `requestMatchers()`

Documentation states that legacy `antMatchers()` have simply been replaced with `requestMatchers()` and then warn that `/foo` matches no longer match `/foo/` URIs. One must explicitly express `/foo` and `/foo/` to make that happen.

```

@Bean
@Order(50)
public SecurityFilterChain h2Configuration(HttpSecurity http) throws Exception {
    http.securityMatchers(cfg->cfg.requestMatchers(①
        "/h2-console/**", "/login", "/logout"));
    ...
}

@Bean
public WebSecurityCustomizer authzStaticResources() {
    return (web) -> web.ignoring().requestMatchers(①
        "/content/**");
}

```

① `requestMatchers()` now defines the match details

In reality, the `requestMatchers()` will resolve to the `mvcMatchers()` when using WebMVC and that is simply how the `mvcMatchers()` work. I assume that is what Spring Security wants you to use. Otherwise the convenient alternate builders would not have been removed or at least the instructions would have more prominently identified how to locate the explicit builders in the new API.

Using antMatcher and regexMatcher

```

import org.springframework.security.web.util.matcher.AntPathRequestMatcher;
import org.springframework.security.web.util.matcher.RegexRequestMatcher;

http.authorizeHttpRequests(cfg->cfg
    .requestMatchers(AntPathRequestMatcher.antMatcher("...")).hasRole("...")
    .requestMatchers(RegexRequestMatcher.regexMatcher("...")).hasRole("...")
    .requestMatchers("/h2-console/**").authenticated()); //MvcRequestMatcher

```

A simple search and replace can be performed for this update as long as `mvcMatchers()` is a suitable solution.

Search/Replace antMatchers with requestMatchers Builder

```

for file in `find . -name "*.java" -exec grep -l 'antMatchers(' {} \;`; do sed -i '' '
's/antMatchers(/requestMatchers(/' $file; done

```

411.3. ignoringAntMatchers/ignoringRequestMatchers

The same is true for the ignoring case. Just replace the `ignoringAntMatchers()` builder method with `ignoringRequestMatchers()`.

Replace ignoringAntMatchers with ignoringRequestMatchers Builder

```

http.csrf(cfg->cfg.ignoringAntMatchers("/h2-console/**"));
...

```

```
http.csrf(cfg->cfg.ignoringRequestMatchers("/h2-console/**"));
```

Search/Replace ignoringAntMatchers with ignoringRequestMatchers Builder

```
for file in `find . -name "*.java" -exec grep -l 'ignoringAntMatchers(' {} \;`; do sed -i '' 's/ignoringAntMatchers(/ignoringRequestMatchers(/' $file; done
```

411.4. authorizeRequests/authorizeHttpRequests

Spring Boot 2.x/Spring 5 used the `authorizeRequests()` builder to define access restrictions for a URI.

```
http.authorizeRequests(cfg->cfg.requestMatchers(  
    "/api/whoami", "/api/authorities/paths/anonymous/**").permitAll());
```

The builder still exists, but has been deprecated for `authorizeHttpRequests()`.

```
http.authorizeHttpRequests(cfg->cfg.requestMatchers(  
    "/api/whoami", "/api/authorities/paths/anonymous/**").permitAll());
```

A simple search and replace can address this issue.

Search/Replace authorizeRequests with authorizeHttpRequests Builder

```
for file in `find . -name "*.java" -exec grep -l 'authorizeRequests(' {} \;`; do sed -i '' 's/authorizeRequests(/authorizeHttpRequests(/' $file; done
```

Chapter 412. Role Hierarchy

Early Spring Security 3.x omission left off automatic support for role inheritance.

412.1. Spring Boot 2.x Role Inheritance

The following shows the seamless integration of role access constraints and role hierarchy definition for security mechanisms that support hierarchies.

```
@Bean
public RoleHierarchy roleHierarchy() {
    RoleHierarchyImpl roleHierarchy = new RoleHierarchyImpl();
    roleHierarchy.setHierarchy(StringUtils.join(List.of(
        "ROLE_ADMIN > ROLE_CLERK",
        "ROLE_CLERK > ROLE_CUSTOMER"
    ), System.lineSeparator())));
    return roleHierarchy;
}
```

```
http.authorizeRequests(cfg->cfg.antMatchers(
    "/api/authorities/paths/customer/**")
    .hasAnyRole("CUSTOMER"));
http.authorizeRequests(cfg->cfg.antMatchers(HttpMethod.GET,
    "/api/authorities/paths/price")
    .hasAnyAuthority("PRICE_CHECK", "ROLE_ADMIN", "ROLE_CLERK"));
```

412.2. Spring Boot 3.x Role Inheritance

The role hierarchies are optionally stored within an AuthorizationManager. Early Spring Boot 3 left that automatic registration out but was available in an up-coming merge request. An [interim solution](#) was to manually supply the `SecurityFilterChain` an `AuthorizationManager` pre-registered with a `RoleHierarchy` definition.

```
http.authorizeHttpRequests(cfg->cfg.requestMatchers(
    "/api/authorities/paths/customer/**")
    .access(anyRoleWithRoleHierarchy(roleHierarchy, "CUSTOMER")))
);
http.authorizeHttpRequests(cfg->cfg.requestMatchers(HttpMethod.GET,
    "/api/authorities/paths/price")
    .access(anyAuthorityWithRoleHierarchy(roleHierarchy, "PRICE_CHECK",
    "ROLE_ADMIN", "ROLE_CLERK")))
);
```

The following snippets show the definition of the `RoleHierarchy` injected into the `SecurityChainFilter` builder. Two have been defined—one for `roleInheritance` profile and one for otherwise.

```

@Bean
@Profile("roleInheritance")
public RoleHierarchy roleHierarchy() {
    RoleHierarchyImpl roleHierarchy = new RoleHierarchyImpl();
    roleHierarchy.setHierarchy(StringUtils.join(List.of(
        "ROLE_ADMIN > ROLE_CLERK",
        "ROLE_CLERK > ROLE_CUSTOMER"
    ), System.lineSeparator())));
    return roleHierarchy;
}

@Bean
@Profile("!roleInheritance")
public RoleHierarchy nullHierarchy() {
    return new NullRoleHierarchy();
}

```

Temporary Builder code to supply Injected Authorization Manager

```

//temporary work-around until this fix is available
//https://github.com/spring-projects/spring-security/issues/12473
private AuthorizationManager anyRoleWithRoleHierarchy(RoleHierarchy roleHierarchy,
String...roles) {
    AuthorityAuthorizationManager<Object> authzManager =
AuthorityAuthorizationManager.hasAnyRole(roles);
    authzManager.setRoleHierarchy(roleHierarchy);
    return authzManager;
}
private AuthorizationManager anyAuthorityWithRoleHierarchy(RoleHierarchy
roleHierarchy, String...authorities) {
    AuthorityAuthorizationManager<Object> authzManager =
AuthorityAuthorizationManager.hasAnyAuthority(authorities);
    authzManager.setRoleHierarchy(roleHierarchy);
    return authzManager;
}

```

Chapter 413. Annotated Method Security

`@EnableGlobalMethodSecurity` has been renamed to `@EnableMethodSecurity` and `prePostEnabled` has been enabled by default.

Spring Boot 2.x @EnableGlobalMethodSecurity

```
@EnableGlobalMethodSecurity(prePostEnabled = true) ①
public class AuthoritiesTestConfiguration {
```

① `prePostEnabled` had to be manually enabled

Spring Boot 3.x @EnableMethodSecurity

```
@EnableMethodSecurity // (prePostEnabled = true) now default
public class AuthoritiesTestConfiguration {
```

A simple search and replace solution should be enough to satisfy the deprecation.

Search/Replace EnableGlobalMethodSecurity with EnableMethodSecurity Annotation

```
for file in `find . -name "*.java" -exec grep -l 'EnableGlobalMethodSecurity(' {} \;`;
do sed -i '' 's@EnableGlobalMethodSecurity(/EnableMethodSecurity(/' $file; done
```

Chapter 414. @Secured

Spring Boot 3.x **@Secured** annotation now supports non-ROLE authorities

```
@Secured({"ROLE_ADMIN", "ROLE_CLERK", "PRICE_CHECK"})
@GetMapping(path = "price", produces = {MediaType.TEXT_PLAIN_VALUE})
public ResponseEntity<String> checkPrice()
```

Chapter 415. JSR250 RolesAllowed

Spring Boot 2.x Jsr250 `ROLE`-s started with the `ROLE_` prefix when defined. Permissions (`PRICE_CHECK`) did not.

415.1. Spring Boot 2.x

Spring Boot 2.x Method Security using @Secured

```
@RolesAllowed("ROLE_CLERK")
public ResponseEntity<String> doClerk()

@RolesAllowed("ROLE_CUSTOMER")
public ResponseEntity<String> doCustomer()

@RolesAllowed({"ROLE_ADMIN", "ROLE_CLERK", "PRICE_CHECK"})
public ResponseEntity<String> checkPrice()
```

415.2. Spring Boot 3.x

Spring Boot 3.x Jsr250 `ROLE`-s definition no longer start with `ROLE_` prefix—just like Permissions (`PRICE_CHECK`).

Spring Boot 3.x Method Security using @Secured

```
@RolesAllowed("CLERK")
public ResponseEntity<String> doClerk()

@RolesAllowed("CUSTOMER")
public ResponseEntity<String> doCustomer()

@RolesAllowed({"ADMIN", "CLERK", "PRICE_CHECK"})
public ResponseEntity<String> checkPrice()
```

Chapter 416. HTTP Client

Lower-level client networking details for `RestTemplate` is addressed using HTTP Client. This primarily includes TLS (still referred to as SSL) but can also include other features like caching and debug logging.

416.1. Spring Boot 2 HTTP Client

Spring Boot 2 used `HttpClient`. The following snippet shows how the TLS could be optionally configured for HTTPS communications.

HttpClient and SSLContext imports

```
import org.apache.http.client.HttpClient;
import org.apache.http.impl.client.HttpClientBuilder;
import org.apache.http.ssl.SSLContextBuilder;
import javax.net.ssl.SSLContext;
```

We first need to establish an `SSLContext` with the definition of protocols and an optional trustStore. The trustStore is optional to communicate with globally trusted sites, but necessary if we communicate using HTTPS with self-generated certs.

The following Spring Boot 2 example, uses an injected definition of the external server to load the trustStore and build the `SSLContext`.

Building Customized SSLContext

```
@Bean
public SSLContext sslContext(ServerConfig serverConfig) {
    try {
        URL trustStoreUrl = null;
        if (serverConfig.getTrustStore() != null) {
            trustStoreUrl = ClientITConfiguration.class.getResource "/" +
serverConfig.getTrustStore());
            if (null==trustStoreUrl) {
                throw new IllegalStateException("unable to locate truststore:/"
serverConfig.getTrustStore());
            }
        }
        SSLContextBuilder builder = SSLContextBuilder.create()
            .setProtocol("TLSv1.2");
        if (trustStoreUrl!=null) {
            builder.loadTrustMaterial(trustStoreUrl, serverConfig
.getTrustStorePassword());
        }
        return builder.build();
    } catch (Exception ex) {
        throw new IllegalStateException("unable to establish SSL context", ex);
    }
}
```

```
}
```

The `SSLContext` and remote server definition are used to build a `HttpClient` to insert into the `ClientHttpRequestFactory` used to establish client connections.

Building Customized ClientHttpRequestFactory with TLS Capability

```
@Bean
public ClientHttpRequestFactory httpsRequestFactory(SSLContext sslContext,
                                                    ServerConfig serverConfig) {
    HttpClient httpsClient = HttpClientBuilder.create()
        .setSSLContext(serverConfig.isHttps() ? sslContext : null)
        .build();
    return new HttpComponentsClientHttpRequestFactory(httpsClient);
}
```

416.2. Spring Boot 3.x HttpClient5

Spring Boot updated the networking in `RestTemplate` to use `httpclient5` and a custom SSL Context library.

HttpClient5 and SSL Factory imports

```
import nl.altindag.ssl.SSLFactory;
import nl.altindag.ssl.util.ApacheSSlUtils;
import org.apache.hc.client5.http.classic.HttpClient;
import org.apache.hc.client5.http.impl.classic.HttpClients;
import org.apache.hc.client5.http.impl.io.PoolingHttpClientConnectionManager;
import org.apache.hc.client5.http.impl.io.PoolingHttpClientConnectionManagerBuilder;
import org.springframework.boot.autoconfigure.condition.ConditionalOnExpression;
```

`httpclient5` uses a `SSLFactory` TLS definition that is similar to its `httpclient` counterpart. The biggest difference in the `@Bean` factory in the example code is that we have decided to disable the bean if the `ServerConfig` `trustStore` property is empty.

Building Customized SSLContext

```
@Bean ①
@ConditionalOnExpression("!T(org.springframework.util.StringUtils).isEmpty('${it.server.trust-store}')")
public SSLFactory sslFactory(ServerConfig serverConfig) throws IOException {
    try (InputStream trustStoreStream = Thread.currentThread()

        .getContextClassLoader().getResourceAsStream(serverConfig.getTrustStore())) {
        if (null==trustStoreStream) {
            throw new IllegalStateException("unable to locate truststore: " +
serverConfig.getTrustStore());
        }
        return SSLFactory.builder()
```

```

        .withProtocols("TLSv1.2")
        .withTrustMaterial(trustStoreStream, serverConfig.getTrustStorePassword())
        .build();
    }
}

```

① SSLFactory will not be created when `it.server.trust-store` is empty

With our design change, we then make the injected `SSLFactory` into the `ClientRequestFactory` @Bean method optional. From there we use `httpclient5` constructs to build the proper components.

Building Customized ClientHttpRequestFactory with TLS Capability

```

@Bean
public ClientHttpRequestFactory httpsRequestFactory(
    @Autowired(required = false) SSLFactory sslFactory) { ①
    PoolingHttpClientConnectionManagerBuilder builder =
        PoolingHttpClientConnectionManagerBuilder.create();
    PoolingHttpClientConnectionManager connectionManager =
        Optional.ofNullable(sslFactory)
            .map(sf -> builder.setSSLSocketFactory(Apache5SslUtils.toSocketFactory(sf
        )))
            .orElse(builder)
            .build();

    HttpClient httpsClient = HttpClient.custom()
        .setConnectionManager(connectionManager)
        .build();
    return new HttpComponentsClientHttpRequestFactory(httpsClient);
}

```

① SSLFactory defined to be optional and checked for null during ConnectionManager creation

Note that `httpclient5` and its TLS extensions require two new dependencies.

httpclient5 and TLS dependencies

```

<dependency>
    <groupId>org.apache.httpcomponents.client5</groupId>
    <artifactId>httpclient5</artifactId>
</dependency>
<dependency>
    <groupId>io.github.hakky54</groupId>
    <artifactId>sslcontext-kickstart-for-apache5</artifactId>
</dependency>

```

The caching extensions `caching extensions` are made available through the following dependency. Take a look at `CachingHttpClientBuilder`.

```
<dependency>
```

```
<groupId>org.apache.httpcomponents.client5</groupId>
<artifactId>httpclient5-cache</artifactId>
</dependency>
```

Chapter 417. Subject Alternative Name (SAN)

Java HTTPS has always had a hostname check that verified the reported hostname matched the DN within the SSL certificate. For local testing, that used to mean only having to supply a `CN=localhost`. Now, the SSL security matches against the subject alternative name ("SAN").

We will get the following error when the service we are calling using HTTPS returns a certificate that does not list a valid subject alternative name (SAN) consistent with the hostname used to connect.

Subject Alternative Name Error

```
ResourceAccess I/O error on GET request for "https://localhost:63848/api/authn/hello":  
Certificate for <localhost> doesn't match any of the subject alternative names: []
```

A valid subject alternative name (SAN) can be generated with the `-ext` parameter within keytool.

Using keytool to supply Subject Alternative Name (SAN)

```
#https://stackoverflow.com/questions/50928061/certificate-for-localhost-doesnt-match-  
any-of-the-subject-alternative-names  
#https://ultimatesecurity.pro/post/san-certificate/  
  
keytool -genkeypair -keyalg RSA -keysize 2048 -validity 3650 \  
-ext "SAN:c=DNS:localhost,IP:127.0.0.1" \①  
-dname "CN=localhost,OU=Unknown,O=Unknown,L=Unknown,ST=Unknown,C=Unknown" \  
-keystore keystore.p12 -alias https-hello \  
-storepass password
```

- ① clients will accept `localhost` or `127.0.0.1` returned from the SSL connection provided by this trusted certificate

Chapter 418. Swagger Changes

418.1. Spring Doc

418.1.1. Spring Boot 2.x

Spring Doc supported Spring Boot 2.x with their 1.x version.

```
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-ui</artifactId>
    <version>1.6.9</version>
</dependency>
```

418.2. Spring Boot 3.x

Spring Doc supports Spring Boot 3.x with their 2.x version.

```
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>2.1.0</version>
</dependency>
```

Chapter 419. JPA Dependencies

419.1. Spring Boot 3.x/Hibernate 6.x

Spring Boot 3.x/Hibernate 6.x requires a dependency on a Validator.

Hibernate 6.x requires Validator

```
jakarta.validation.NoProviderFoundException: Unable to create a Configuration, because no Jakarta Bean Validation provider could be found. Add a provider like Hibernate Validator (RI) to your classpath.
```

To correct, add the validation starter.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

Chapter 420. JPA Default Sequence

One mechanism for generating a primary key value is to use a sequence.

420.1. Spring Boot 2.x/Hibernate 5.x

Spring Boot 2.x/Hibernate 5.x used to default the sequence to `hibernate_sequence`.

Example Sequence Generator Definition without a Name

```
@Entity  
public class Song {  
    @Id @GeneratedValue(strategy = GenerationType.SEQUENCE)  
    private int id;
```

Spring Boot2.x/Hibernate 5.x Defaulted to hibernate_sequence

```
enum Dialect {  
    H2("call next value for hibernate_sequence"),  
    POSTGRES("select nextval('hibernate_sequence')");
```

```
drop sequence IF EXISTS hibernate_sequence;  
create sequence hibernate_sequence start with 1 increment 1;
```

420.2. Spring Boot 3.x/Hibernate 6.x

Spring Boot 3.x/Hibernate 6.x no longer permit an unnamed sequence generator. It must be named.



Default increment has changed

The default increment for the sequence has also changed from 1 to 50.

Example Named Sequence Generator Definition

```
@Entity  
public class Song {  
    @Id @GeneratedValue(strategy = GenerationType.SEQUENCE, generator =  
    "REPOSONGS_SONG_SEQUENCE")  
    private int id;
```

Spring Boot3.x/Hibernate 6.x with Named Sequence

```
enum Dialect {  
    H2("call next value for REPOSONGS_SONG_SEQUENCE"),  
    POSTGRES("select nextval('REPOSONGS_SONG_SEQUENCE')");
```

```
drop sequence IF EXISTS REPOSONGS_SONG_SEQUENCE;  
create sequence REPOSONGS_SONG_SEQUENCE start with 1 increment 50; ①
```

① default increment is 50

Chapter 421. JPA Property Changes

421.1. Spring Boot 2.x/Hibernate 5.x

The legacy JPA persistence properties used a `javax` prefix.

Spring Boot 2.x/Hibernate 5.x JPA javax Property Prefix

```
spring.jpa.properties.javaee.persistence.schema-generation.create-source=metadata  
spring.jpa.properties.javaee.persistence.schema-generation.scripts.action=drop-and-create  
spring.jpa.properties.javaee.persistence.schema-generation.scripts.create-target=target/generated-sources/ddl/drop_create.sql  
spring.jpa.properties.javaee.persistence.schema-generation.scripts.drop-target=target/generated-sources/ddl/drop_create.sql
```

Hibernate moved some classes. `org.hibernate.type` logging changed to `org.hibernate.orm.jdbc.bind` in later versions.

Hibernate 5.x Bind Property Logging

```
logging.level.org.hibernate.type=TRACE
```

421.2. Spring Boot 3.x/Hibernate 6.x

With Spring Boot 3.x/Hibernate 6.x, the property prefix has changed to `jakarta`.

Spring Boot 3.x/Hibernate 6.x JPA jakarta Property Prefix

```
spring.jpa.properties.jakarta.persistence.schema-generation.create-source=metadata  
spring.jpa.properties.jakarta.persistence.schema-generation.scripts.action=drop-and-create  
spring.jpa.properties.jakarta.persistence.schema-generation.scripts.create-target=target/generated-sources/ddl/drop_create.sql  
spring.jpa.properties.jakarta.persistence.schema-generation.scripts.drop-target=target/generated-sources/ddl/drop_create.sql
```

Hibernate 6.x Bind Property Logging

```
logging.level.org.hibernate.orm.jdbc.bind=TRACE
```

Chapter 422. Embedded Mongo

422.1. Embedded Mongo AutoConfiguration

Spring Boot 3.x has removed direct support for Flapdoodle in favor of configuring it yourself or using [testcontainers](#).

The legacy [EmbeddedMongoAutoConfiguration](#) can now be found in a flapdoodle package.

Spring Boot 2.x Flapdoodle AutoConfiguration

```
import  
org.springframework.boot.autoconfigure.mongo.embedded.EmbeddedMongoAutoConfiguration;
```

Spring Boot 3.x Flapdoodle AutoConfiguration

```
import de.flapdoodle.embed.mongo.spring.autoconfigure.EmbeddedMongoAutoConfiguration;
```

422.2. Embedded Mongo Properties

The mandatory [mongodb.embedded.version](#) property has been renamed

- from: [spring.mongodb.embedded.version](#)
- to: [de.flapdoodle.mongodb.embedded.version](#)

It works exactly the same as before.

Spring Boot 2.x Mandatory Flapdoodle Property

```
spring.mongodb.embedded.version=4.4.0
```

Spring Boot 3.x Mandatory Flapdoodle Property

```
de.flapdoodle.mongodb.embedded.version=4.4.0
```

A simple search and replace of property files addresses this change. YAML file changes would have been more difficult.

```
for file in `find . -name "*.properties" -exec egrep -l  
'spring.mongodb.embedded.version' {} \;`; do sed -i ''  
's/spring.mongodb.embedded.version/de.flapdoodle.mongodb.embedded.version/' $file;  
done
```

Chapter 423. ActiveMQ/Artemis

ActiveMQ and Artemis are two branches within the ActiveMQ baseline. I believe Artemis came from a JBoss background. Only Artemis has been updated to support [jakarta.jms](#) constructs.

423.1. Spring Boot 2.x

The following snippet shows the Maven dependency for ActiveMQ, with its [jakarta.jms](#) support.

ActiveMQ Dependency

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-activemq</artifactId>
</dependency>
```

The following snippet shows a few example ActiveMQ properties used in the simple example within the course tree.

Example ActiveMQ Properties

```
spring.activemq.broker-url=tcp://activemq:61616
spring.activemq.in-memory=true
spring.activemq.pool.enabled=false
```

423.2. Spring Boot 3.x

The following snippet shows the Maven dependencies for Artemis and its [jakarta.jms](#) support. The Artemis server dependency had to be separately added in order to run an embedded JMS server. JMSTemplate also recommended the Pooled JMS dependency to tune the use of connections.

Artemis Dependencies

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-artemis</artifactId>
</dependency>
<!-- jmsTemplate connection polling -->
<dependency>
    <groupId>org.messaginghub</groupId>
    <artifactId>pooled-jms</artifactId>
</dependency>
<!-- dependency added a runtime server to allow running with embedded topic -->
<dependency>
    <groupId>org.apache.activemq</groupId>
    <artifactId>artemis-jakarta-server</artifactId>
```

```
</dependency>
```

The following snippet shows a few example Artemis properties and configuring the JMS connection pool.

Artemis Replacement Properties

```
spring.artemis.broker-url=tcp://activemq:61616  
  
#requires org.messaginghub:pooled-jms dependency  
#https://activemq.apache.org/spring-support  
spring.artemis.pool.enabled=true  
spring.artemis.pool.max-connections=5
```

Chapter 424. Summary

In this module we:

- Identified dependency definition and declaration changes between Spring Boot 2.x and 3.x
- Identified code changes required to migrate from Spring Boot 2.x to 3.x

With a Spring Boot 3/Spring 6 baseline, we can now move forward with some of the latest changes in the Spring Boot ecosystem.

Chapter 425. Porting to Spring Boot 3.3.2

425.1. JarLauncher

Spring Boot changed the Java package name for the JarLauncher.

- from: `org.springframework.boot.loader.JarLauncher`
- to: `org.springframework.boot.loader.launch.JarLauncher`

This is the class that invokes our main(). This primarily has an impact to Dockerfiles

JarLauncher Moved to Different Package

```
#ENTRYPOINT ["/run_env.sh", "java","org.springframework.boot.loader.JarLauncher"]  
  
#https://github.com/spring-projects/spring-boot/issues/37667  
ENTRYPOINT ["/run_env.sh",  
"java","org.springframework.boot.loader.launch.JarLauncher"]
```

It is also visible in the MANIFEST.MF

JarLauncher Bootstraps Application as Main-Class

```
$ unzip -p target/springboot-app-example-*~SNAPSHOT-bootexec.jar META-INF/MANIFEST.MF  
  
Manifest-Version: 1.0  
Created-By: Maven JAR Plugin 3.3.0  
Build-Jdk-Spec: 17  
Main-Class: org.springframework.boot.loader.launch.JarLauncher  
Start-Class: info.ejava.examples.app.build.springboot.SpringBootApp  
Spring-Boot-Version: 3.3.2  
Spring-Boot-Classes: BOOT-INF/classes/  
Spring-Boot-Lib: BOOT-INF/lib/  
Spring-Boot-Classpath-Index: BOOT-INF/classpath.idx  
Spring-Boot-Layers-Index: BOOT-INF/layers.idx
```

425.2. Role Inheritance

@Secured and Jsr250 (@RolesAllowed) now again support Role inheritance. Role Inheritance has also gained a first class builder versus the legacy XML-ish approach.

The following shows the basic syntax of how to declare each role restriction.

Role Authorization Requirement Declarations

```
@PreAuthorize("hasRole('CUSTOMER')")  
@GetMapping(path = "customer", produces = {MediaType.TEXT_PLAIN_VALUE})  
public ResponseEntity<String> doCustomer( ...
```

```

//@Secured-based
@Secured("ROLE_CUSTOMER")
@GetMapping(path = "customer", produces = {MediaType.TEXT_PLAIN_VALUE})
public ResponseEntity<String> doCustomer( ...)

//Jsr250-based
@RolesAllowed("CUSTOMER")
@GetMapping(path = "customer", produces = {MediaType.TEXT_PLAIN_VALUE})
public ResponseEntity<String> doCustomer( ...)

//path-based
@GetMapping(path = "customer", produces = {MediaType.TEXT_PLAIN_VALUE})
public ResponseEntity<String> doCustomer( ...)

@Bean
public SecurityFilterChain authzSecurityFilters(HttpSecurity http,
    MvcRequestMatcher.Builder mvc) throws Exception {
    http.authorizeHttpRequests(cfg->cfg.requestMatchers(
        mvc.pattern("/api/authorities/paths/customer/**"))
        .hasAnyRole("CUSTOMER")));
    ...
}

```

The following is how we defined the role inheritance—which is now honored by @Secured and Jsr250.

Legacy Role Inheritance Definition

```

@Bean
@Profile("roleInheritance")
static RoleHierarchy roleHierarchy() {
    RoleHierarchyImpl roleHierarchy = new RoleHierarchyImpl();
    roleHierarchy.setHierarchy(StringUtils.join(List.of(
        "ROLE_ADMIN > ROLE_CLERK",
        "ROLE_CLERK > ROLE_CUSTOMER"
    ), System.lineSeparator()));
    return roleHierarchy;
}

```

The new upgrades supply a new builder for the `RoleHierarchy` versus the klunky XML-ish legacy approach.

New Role Inheritance Builder

```

@Bean
@Profile("roleInheritance")
static RoleHierarchy roleHierarchy() {
    return RoleHierarchyImpl.withDefaultRolePrefix()
        .role("ADMIN").implies("CLERK")
        .role("CLERK").implies("CUSTOMER")
}

```

```
        .build();  
    }  
}
```

425.3. Flyway 10: Unsupported Database: PostgreSQL

Flyway 10 has refactored their JARs such that individual database support, with the exception of H2, no longer comes from the flyway-core dependency. The following snippet shows the error when running against PostgreSQL (any version).

```
21:07:06.127 main  WARN o.s.w.c.s.GenericWebApplicationContext#refresh:633 Exception  
encountered during context initialization - cancelling refresh attempt:  
org.springframework.beans.factory.BeanCreationException: Error creating bean with name  
'entityManagerFactory' defined in class path resource  
[org/springframework/boot/autoconfigure/orm/jpa/HibernateJpaConfiguration.class]:  
Failed to initialize dependency 'flywayInitializer' of LoadTimeWeaverAware bean  
'entityManagerFactory': Error creating bean with name 'flywayInitializer' defined in  
class path resource  
[org/springframework/boot/autoconfigure/flyway/FlywayAutoConfiguration$FlywayConfigura  
tion.class]: Unsupported Database: PostgreSQL 12.3
```

To correct the issue, add an extra dependency on [org.flywaydb:flyway-database-postgresql](#) or [whatever your specific database requires](#).

```
<dependency>  
    <groupId>org.flywaydb</groupId>  
    <artifactId>flyway-core</artifactId>  
    <scope>runtime</scope>  
</dependency>  
<dependency>  
    <groupId>org.flywaydb</groupId>  
    <artifactId>flyway-database-postgresql</artifactId>  
    <scope>runtime</scope>  
</dependency>
```

H2 support is still bundled within [flyway-core](#), so there is no [flyway-database-h2](#) dependency to add.

425.4. MongoHealthIndicator

Spring Boot added actuator UP/DOWN health support for MongoDB. However, the command it issues to determine if the server is UP is not supported by all versions and not supported by the [4.4.0-bionic](#) typically used for class. This results in the [/actuator/health](#) endpoint to report the overall application as DOWN. Several of the pre-integration-test phases rely on the [/actuator/health](#) endpoint as a generic test for the application to have started and finished initialization.

There are two or more solutions

1. turn off the MongoHealthIndicator

```
#MongoHealthIndicator is not compatible with 4.4.0.  
#https://github.com/spring-projects/spring-boot/issues/41101  
#https://stackoverflow.com/questions/41803253/application-status-down-when-mongo-  
is-down-with-spring-boot-actuator  
  
management.health.mongo.enabled=false
```

2. upgrade the MongoDB to a supported version

```
# or use more modern version of mongodb to support MongoHealthIndicator  
#https://github.com/spring-projects/spring-boot/issues/41101  
#docker-compose  
services:  
  mongodb:  
    #      image: mongo:4.4.0-bionic  
    image: mongo:4.4.28
```

3. override the MongoHealthIndicator bean with your own definition

I have successfully tested with a later version of MongoDB but chose to turn off the feature until I have time to address a MongoDB and PostgresSQL upgrade together across all projects.

425.5. Spring MVC @Validated

The Spring @Validated annotation is no longer required to trigger Jakarta validation.

The way I found it was a little esoteric—driven by a class example looking to demonstrate validation in the controller versus service, using the same implementation classes, and configure them using inheritance overrides during profile-based `@Bean` construction.

Legacy Spring MVC would not trigger Jakarta validation without a Spring @Validated annotation at the class level. Now, without the presence of Spring's @Validated, Spring MVC will automatically activate Jakarta validation in the controller and throw `MethodArgumentNotValidException` exception when violated.

Jakarta Validation Constraint without @Validated

```
@RestController  
//no @Validated annotation here  
public class ContactsController extends ContactsController {  
    @RequestMapping(path= CONTACT_PATH,  
                    method=RequestMethod.GET,  
                    produces={MediaType.APPLICATION_JSON_VALUE, MediaType  
.APPLICATION_XML_VALUE})  
    public ResponseEntity<PersonPocDTO> createPOC(  
        @Validated(PocValidationGroups.CreatePlusDefault.class)
```

```
    @RequestBody PersonPocDTO personDTO) {  
        return super.createPOC(personDTO);  
    }  
}
```

With legacy Spring MVC, the above controller would not trigger validation unless annotated with `@Validated` using a derived class.

Validation Explicitly Activated

```
@RestController  
@Validated  
public class ValidatingContactsController extends ContactsController {
```

The way I was able to restore the override (i.e., turn off Jakarta validation in the base controller class) was to set the validated profile to something that did not exist.

Turning Off Automatic Jakarta Validation

```
static interface NoValidation {}  
  
@Validated(NoValidation.class) //turning off default activation  
public class NonValidatingContactsController extends ContactsController {  
    public ResponseEntity<PersonPocDTO> createPOC(  
        @Validated(NoValidation.class) //overriding parent with non-existant group  
        PersonPocDTO personDTO) {  
            return super.createPOC(personDTO);  
    }  
}
```

This was class example-driven, but it does mean that the need for `@Validated` to trigger Jakarta validation no longer exists in Spring MVC.

JWT/JWS Token Authn/Authz

copyright © 2024 jim stafford (jim.stafford@jhu.edu)

Chapter 426. Introduction

In previous lectures we have covered many aspects of the Spring/Spring Boot authentication and authorization frameworks and have mostly demonstrated that with HTTP Basic Authentication. In this lecture we are going to use what we learned about the framework to implement a different authentication strategy—JSON Web Token (JWT) and JSON Web Signature (JWS).

The focus on this lecture will be a brief introduction to JSON Web Tokens (JWT) and how they could be implemented in the Spring/Spring Boot Security Framework. The real meat of this lecture is to provide a concrete example of how to leverage and extend the provided framework.

426.1. Goals

You will learn:

- what is a JSON Web Token (JWT) and JSON Web Secret (JWS)
- what problems does JWT/JWS solve with API authentication and authorization
- how to write and integrate custom authentication and authorization framework classes to implement an alternate security mechanism
- how to leverage Spring Expression Language to evaluate parameters and properties of the `SecurityContext`

426.2. Objectives

At the conclusion of this lecture and related exercises, you will be able to:

1. construct and sign a JWT with claims representing an authenticated user
2. verify a JWS signature and parse the body to obtain claims to re-instantiate an authenticated user details
3. identify the similarities and differences in flows between HTTP Basic and JWS authentication/authorization flows
4. build a custom JWS authentication filter to extract login information, authenticate the user, build a JWS bearer token, and populate the HTTP response header with its value
5. build a custom JWS authorization filter to extract the JWS bearer token from the HTTP request, verify its authenticity, and establish the authenticated identity for the current security context
6. implement custom error reporting with authentication and authorization

Chapter 427. Identity and Authorities

Some key points of security are to identify the caller and determine authorities they have.

- When using BASIC authentication, we presented credentials each time. This was all in one shot, every time on the way to the operation being invoked.
- When using FORM authentication, we presented credentials (using a FORM) up front to establish a session and then referenced that session on subsequent calls.

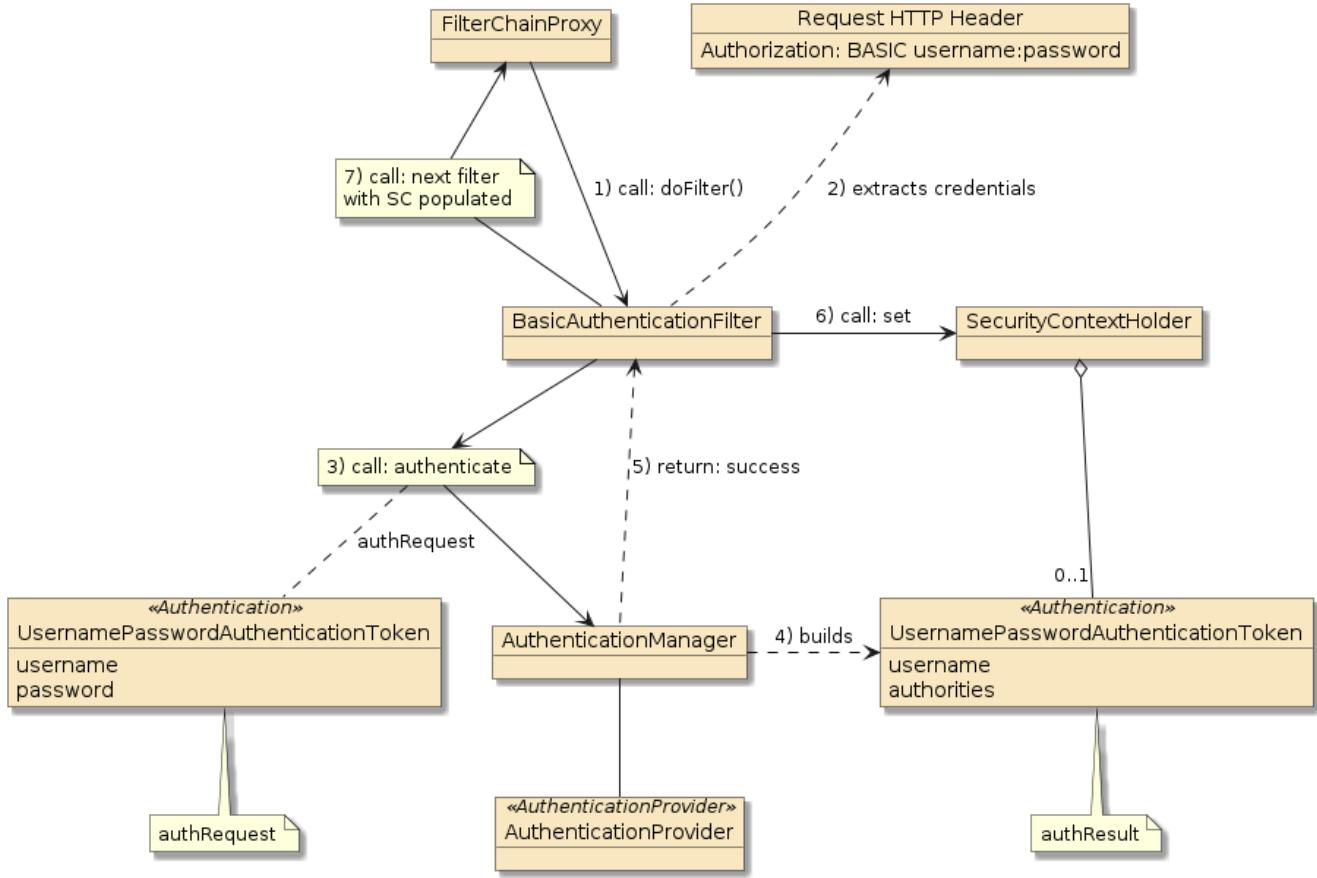
The benefit to BASIC is that it is stateless and can work with multiple servers—whether clustered or peer services. The bad part about BASIC is that we must present the credentials each time and the services must have access to our user details (including passwords) to be able to do anything with them.

The benefit to FORM is that we present credentials one time and then reference the work of that authentication through a session ID. The bad part of FORM is that the session is on the server and harder to share with members of a cluster and impossible to share with peer services.

What we intend to do with token-based authentication is to mimic the one-time login of FORM and stateless aspects of BASIC. To do that—we must give the client at login, information they can pass to the services hosting operations that can securely identify them (at a minimum) and potentially identify the authorities they have without having that stored on the server hosting the operation.

427.1. BASIC Authentication/Authorization

To better understand the token flow, I would like to start by reviewing the BASIC Auth flow.



1. the **BasicAuthenticationFilter** ("the filter") is called in its place within the **FilterChainProxy**
2. the filter extracts the username/password credentials from the **Authorization** header and stages them in a **UsernamePasswordAuthenticationToken** ("the authRequest")
3. the filter passes the authRequest to the **AuthenticationManager** to authenticate
4. the **AuthenticationManager**, thru its assigned **AuthenticationProvider**, successfully authenticates the request and builds an authResult
5. the filter receives the successful response with the authResult hosting the user details—including username and granted authorities
6. the filter stores the authResult in the **SecurityContext**
7. the filter invokes the next filter in the chain—which will eventually call the target operation

All this—authentication and user details management—must occur within the same server as the operation for BASIC Auth.

Chapter 428. Tokens

With token authentication, we are going to break the flow into two parts: authentication/login and authorization/operation.

428.1. Token Authentication/Login

The following is a conceptual depiction of the authentication flow. It differs from the BASIC Authentication flow in that nothing is stored in the **SecurityContext** during the login/authentication. Everything needed to authorize the follow-on operation call is encoded into a **Bearer Token** and returned to the caller in an **Authorization** header. Things encoded in the bearer token are referred to as "claims".

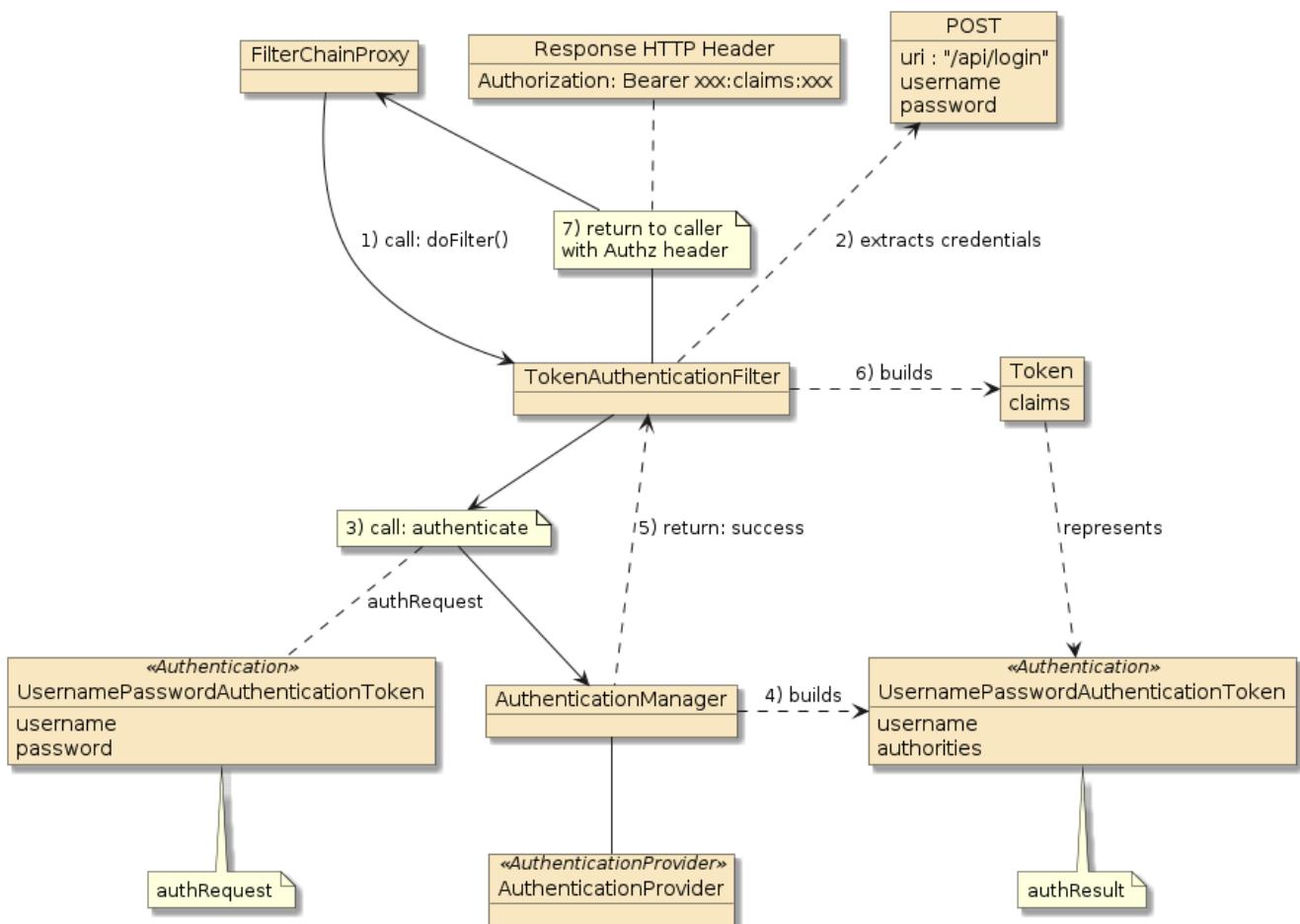


Figure 181. Example Notional Token Authentication/Login

Step 2 extracts the username/password from a POST payload—very similar to FORM Auth. However, we could have just as easily implemented the same extract technique used by BASIC Auth.

Step 7 returns the the token representation of the authResult back to the caller that just successfully authenticated. They will present that information later when they invoke an operation in this or a different server. There is no requirement for the token returned to be used locally. The token can be used on any server that trusts tokens created by this server. The biggest requirement is that we must trust the token is built by something of trust and be able to verify that it never gets modified.

428.2. Token Authorization/Operation

To invoke the intended operation, the caller must include an **Authorization** header with the bearer token returned to them from the login. This will carry their identity (at a minimum) and authorities encoded in the bearer token's claims section.

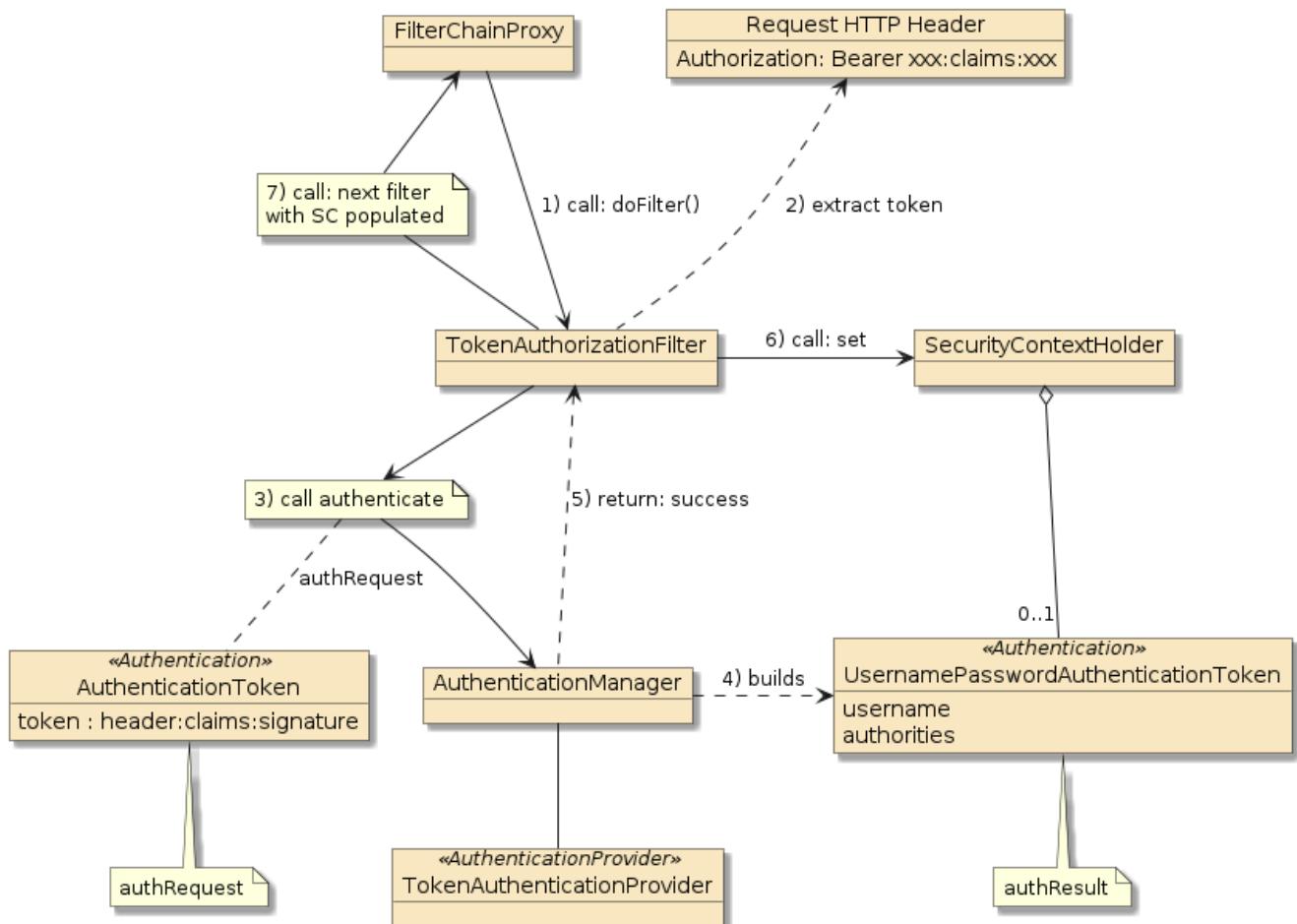


Figure 182. Example Notational Token Authorization/Operation

1. the Token AuthorizationFilter ("the filter") is called by the **FilterChainProxy**
2. the filter extracts the bearer token from the **Authorization** header and wraps that in an **authRequest**
3. the filter passes the **authRequest** to the **AuthenticationManager** to authenticate
4. the **AuthenticationManager** with its Token **AuthenticationProvider** are able to verify the contents of the token and re-build the necessary portions of the **authResult**
5. the **authResult** is returned to the filter
6. the filter stores the **authResult** in the **SecurityContext**
7. the filter invokes the next filter in the chain — which will eventually call the target operation

Bearer Token has Already Been Authenticated

Since the filter knows this is a bearer token, it could have bypassed the call to the **AuthenticationManager**. However, by doing so—it makes the responsibilities of the classes consistent with their original purpose and also gives the



`AuthenticationProvider` the option to obtain more user details for the caller.

428.3. Authentication Separate from Authorization

Notice the overall client to operation call was broken into two independent workflows. This enables the client to present their credentials a limited amount of times and for the operations to be spread out through the network. The primary requirement to allow this to occur is **TRUST**.

We need the ability for the authResult to be represented in a token, carried around by the caller, and presented later to the operations with the trust that it was not modified.

JSON Web Tokens (JWT) are a way to express the user details within the body of a token. JSON Web Signature (JWS) is a way to assure that the original token has not been modified. JSON Web Encryption (JWE) is a way to assure the original token stays private. This lecture and example will focus in JWS—but it is common to refer to the overall topic as JWT.

428.4. JWT Terms

The following table contains some key, introductory terms related to JWT.

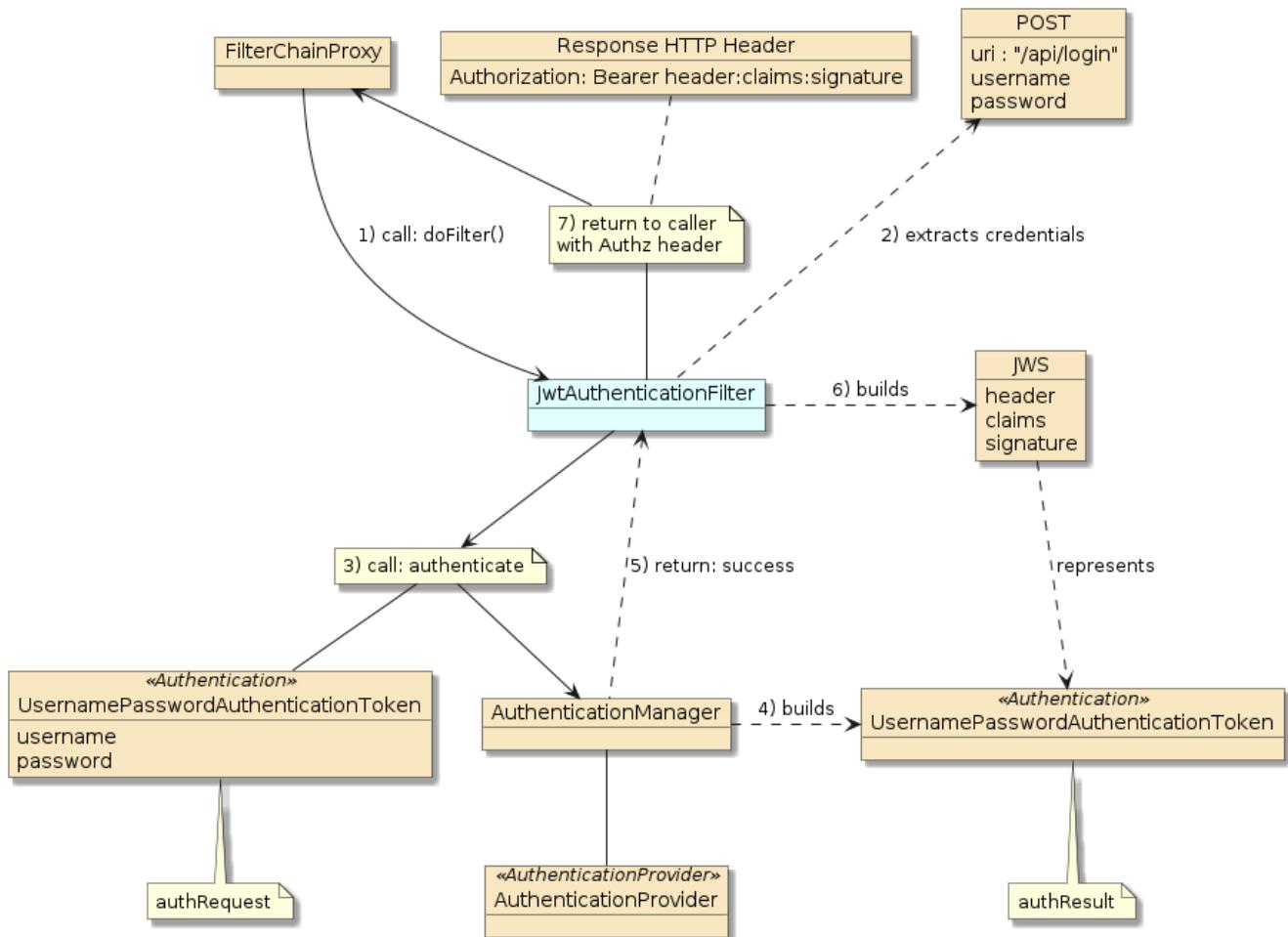
JSON Web Token (JWT)	a compact JSON claims representation that makes up the payload of a JWS or JWE structure e.g., <code>{"sub": "user1", "auth": ["ROLE_ADMIN"]}</code> . The JSON document is referred to as the JWT Claim Set. Basically—this is where we place what we want to represent. In our case, we will be representing the authenticated principal and their assigned authorities.
JSON Web Signature (JWS)	represents content secured with a digital signature (signed with a private key and verifiable using a sharable public key) or Message Authentication Codes (MACs) (signed and verifiable using a shared, symmetric key) using JSON-based data structures
JSON Web Encryption (JWE)	represents encrypted content using JSON-based data structures
JSON Web Algorithms (JWA)	a registry of required, recommended, and optional algorithms and identifiers to be used with JWS and JWE
JSON Object Signing and Encryption (JOSE) Header	JSON document containing cryptographic operations/parameters used. e.g., <code>{"typ": "JWT", "alg": "HS256"}</code>
JWS Payload	the message to be secured—an arbitrary sequence of octets
JWS Signature	digital signature or MAC over the header and payload
Unsecured JWS	JWS without a signature (<code>"alg": "none"</code>)

JWS Compact Serialization	a representation of the JWS as a compact, URL-safe String meant for use in query parameters and HTTP headers <pre>base64({"typ":"JWT", "alg":"HS256"}) .base64({"sub":"user1", "auth":["ROLE_ADMIN"]}) .base64(signature(JOSE + Payload))</pre>
JWS JSON Serialization	a JSON representation where individual fields may be signed using one or more keys. There is no emphasis for compact for this use but it makes use of many of the underlying constructs of JWS.

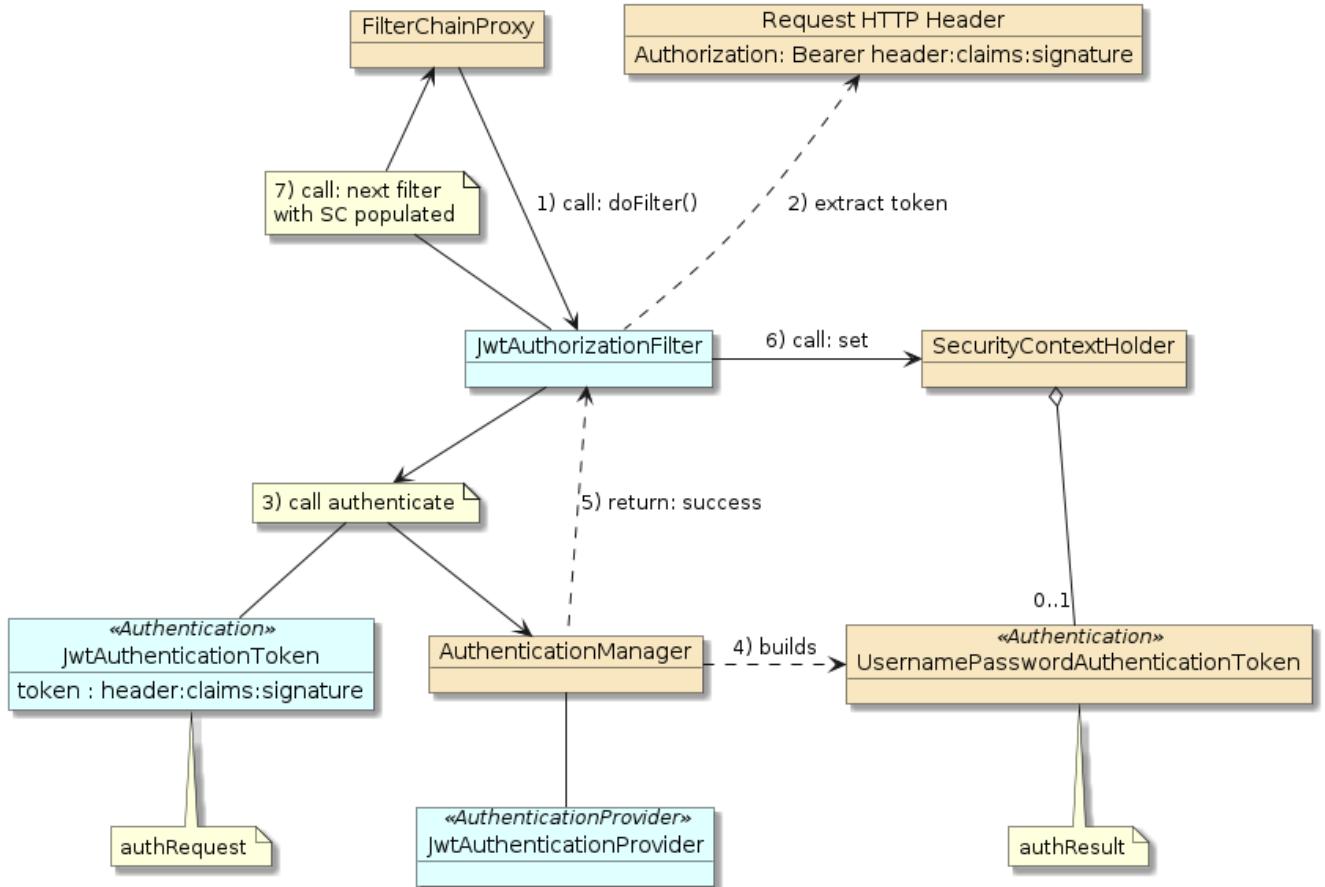
Chapter 429. JWT Authentication

With the general workflows understood and a few concepts of JWT/JWS introduced, I want to update the diagrams slightly with real classnames from the examples and walk through how we can add JWT authentication to Spring/Spring Boot.

429.1. Example JWT Authentication/Login Flow



429.2. Example JWT Authorization/Operation Call Flow



Lets take a look at the implementation to be able to fully understand both JWT/JWS and leveraging the Spring/Spring Boot Security Framework.

Chapter 430. Maven Dependencies

Spring does not provide its own standalone JWT/JWS library or contain a direct reference to any. I happen to be using the [jjwt library from jsonwebtoken](#).

JWT/JWS Maven Dependencies

```
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-api</artifactId>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-impl</artifactId>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-jackson</artifactId>
    <scope>runtime</scope>
</dependency>
```

Chapter 431. JwtConfig

At the bottom of the details of our JWT/JWS authentication and authorization example is a `@ConfigurationProperties` class to represent the configuration.

Example JwtConfig @ConfigurationProperties Class

```
@ConfigurationProperties(prefix = "jwt")
@Data
@Slf4j
public class JwtConfig {
    @NotNull
    private String loginUri; ①
    private String key; ②
    private String authoritiesKey = "auth"; ③
    private String headerPrefix = "Bearer "; ④
    private int expirationSecs=60*60*24; ⑤

    public String getKey() {
        if (key==null) {
            key=UUID.randomUUID().toString();
            log.info("generated JWT signing key={}",key);
        }
        return key;
    }
    public SecretKey getSigningKey() {
        return Keys.hmacShaKeyFor(getKey().getBytes(Charset.forName("UTF-8")));
    }
    public SecretKey getVerifyKey() {
        return getSigningKey();
    }
}
```

① `login-uri` defines the URI for the JWT authentication

② `key` defines a value to build a symmetric `SecretKey`

③ `authorities-key` is the JSON key for the user's assigned authorities within the JWT body

④ `header-prefix` defines the prefix in the `Authorization` header. This will likely never change, but it is good to define it in a single, common place

⑤ `expiration-secs` is the number of seconds from generation for when the token will expire. Set this to a low value to test expiration and large value to limit login requirements

431.1. JwtConfig application.properties

The following shows an example set of properties defined for the `@ConfigurationProperties` class.

Example property value

①

```
jwt.key=12345678901234567890123456789012345678901234567890  
jwt.expiration-secs=300000000  
jwt.login-uri=/api/login
```

- ① the **key** must remain protected—but for symmetric keys must be shared between signer and verifiers

Chapter 432. JwtUtil

This class contains all the algorithms that are core to implementing token authentication using JWT/JWS. It is configured by value in `JwtConfig`.

Example JwtUtil Utility Class

```
@RequiredArgsConstructor  
public class JwtUtil {  
    private final JwtConfig jwtConfig;
```

432.1. Dependencies on JwtUtil

The following diagram shows the dependencies on `JwtUtil` and also on `JwtConfig`.

- `JwtAuthenticationFilter` needs to process requests to the `loginUri`, generate a JWS token for successfully authenticated users, and set that JWS token on the HTTP response
- `JwtAuthorizationFilter` processes all messages in the chain and gets the JWS token from the `Authorization` header.
- `JwtAuthenticationProvider` parses the String token into an `Authentication` result.

`JwtUtil` handles the meat of that work relative to JWS. The other classes deal with plugging that work into places in the security flow.

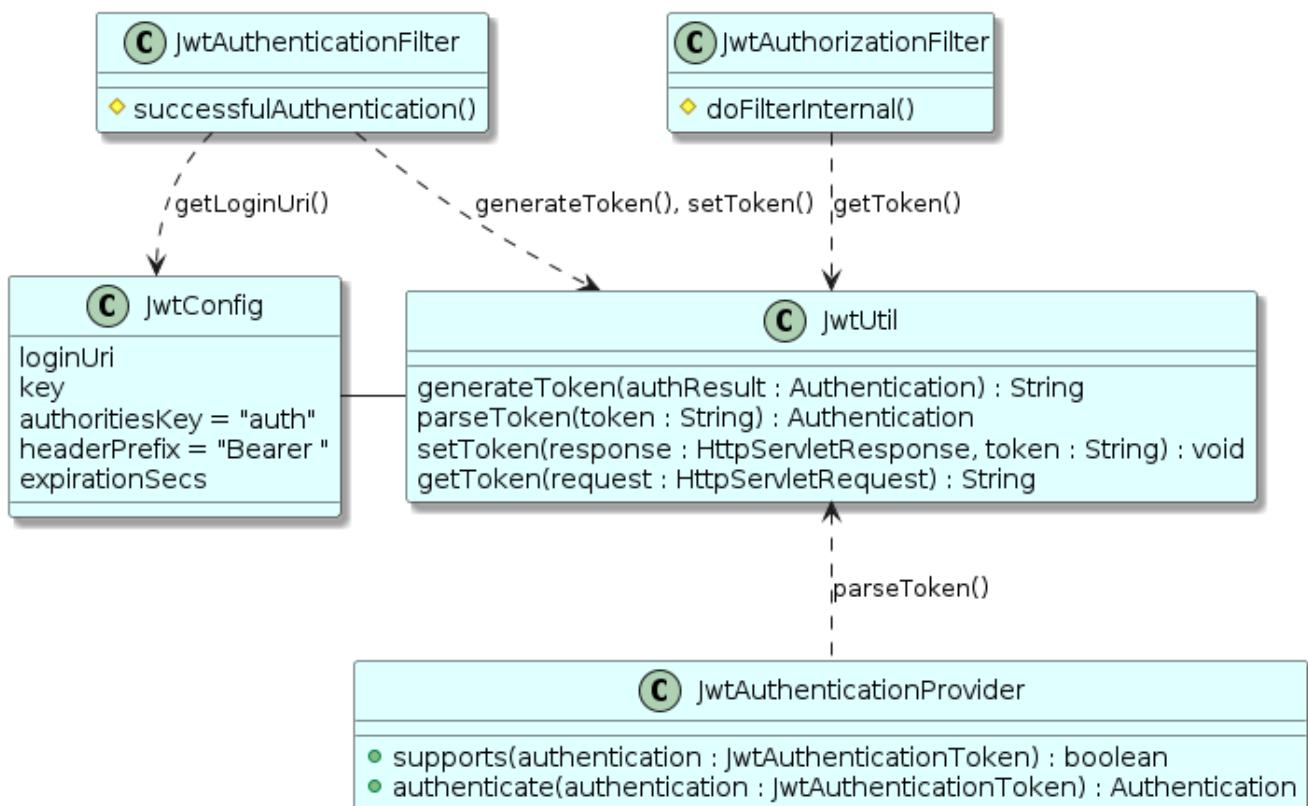


Figure 183. Dependencies on `JwtUtil`

432.2. JwtUtil: generateToken()

The following code snippet shows creating a JWS builder that will end up signing the header and payload. Individual setters are called for well-known claims. A generic claim(key, value) is used to add the authorities.

JwtUtil generateToken() for Authenticated User

```
import io.jsonwebtoken.Jwts;
...
public String generateToken(Authentication authenticated) {
    String token = Jwts.builder()
        .setSubject(authenticated.getName()) ①
        .setIssuedAt(new Date())
        .setExpiration(getExpires()) ②
        .claim(jwtConfig.getAuthoritiesKey(), getAuthorities(authenticated))
        .signWith(jwtConfig.getSigningKey())
        .compact();
    return token;
}
```

① JWT has some well-known claim values

② `claim(key, value)` used to set custom claim values

432.3. JwtUtil: generateToken() Helper Methods

The following helper methods are used in setting the claim values of the JWT.

JwtUtil generateToken() Helper Methods

```
protected Date getExpires() { ①
    Instant expiresInstant = LocalDateTime.now()
        .plus(jwtConfig.getExpirationSecs(), ChronoUnit.SECONDS)
        .atZone(ZoneOffset.systemDefault())
        .toInstant();
    return Date.from(expiresInstant);
}
protected List<String> getAuthorities(Authentication authenticated) {
    return authenticated.getAuthorities().stream() ②
        .map(a->a.getAuthority())
        .collect(Collectors.toList());
}
```

① calculates an instant in the future — relative to local time — the token will expire

② strip authorities down to String authorities to make marshalled value less verbose

The following helper method in the `JwtConfig` class generates a `SecretKey` suitable for signing the JWS.

JwtConfig getSigningKey() Helper Method

```
...
import io.jsonwebtoken.security.Keys;
import javax.crypto.SecretKey;

public class JwtConfig {
    public SecretKey getSigningKey() {
        return Keys.hmacShaKeyFor(getKey() ①
            .getBytes(Charset.forName("UTF-8")));
    }
}
```

① the `hmacSha` algorithm and the 40 character key will generate a `HS384 SecretKey` for signing

432.4. Example Encoded JWS

The following is an example of what the token value will look like. There are three base64 values separated by a period "." each. The first represents the header, the second the body, and the third the cryptographic signature of the header and body.

Example Encoded JWS

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJmcmFzaWVyIiwiaWF0IjoxNTk0ODk1Nzk3LCJle
HAIoJE10TQ40Tk1MTcsImF1dGhvcmI0aWVzIjpbIlBSSUNFX0NIRUNLIiwiUk9MRV9DVVNUT01FUijdLCJqdGk
i0iI5NjQ3MzE1OS03MTNjLTRlN2EtYmE4Zi0zYWMyMzlmODhjZGQifQ.ED-j7md02bwNdZdI4I2Hm_88j-
aSeYkrbd1EacmjotU
```

①

① `base64(JWS Header).base64(JWS body).base64(sign(header + body))`



There is no set limit to the size of HTTP headers. However, it has been [pointed out that Apache defaults to an 8KB limit and IIS is 16KB](#). The default size for [Tomcat is 4KB](#). In case you were counting, the above string is 272 characters long.

432.5. Example Decoded JWS Header and Body

Example Decoded JWS Header and Body

```
{
  "typ": "JWT",
  "alg": "HS384"
}
{
  "sub": "frasier",
  "iat": 1594895797,
  "exp": 1894899397,
  "auth": [
    "PRICE_CHECK",
    "ROLE_CUSTOMER"
  ]
}
```

The following is what is produced if we base64 decode the first two sections. We can use sites like jsonwebtoken.io and jwt.io to inspect JWS tokens. The header identifies the type and signing algorithm. The body carries the claims. Some claims (e.g., subject/`sub`) are well known and standardized. All standard claims are shortened to try to make the token as condensed as possible.

432.6. JwtUtil: `parseToken()`

The `parseToken()` method verifies the contents of the JWS has not been modified, and re-assembles an authenticated `Authentication` object to be returned by the `AuthenticationProvider` and `AuthenticationManager` and placed into the `SecurityContext` for when the operation is executed.

Example JwtUtil `parseToken()`

```
...
import io.jsonwebtoken.Claims;
import io.jsonwebtoken.JwtException;
import io.jsonwebtoken.Jwts;

public Authentication parseToken(String token) throws JwtException {
    Claims body = Jwts.parserBuilder()
        .setSigningKey(jwtConfig.getVerifyKey()) ①
        .build()
        .parseClaimsJws(token)
        .getBody();
    User user = new User(body.getSubject(), "", getGrantedAuthorities(body));
    Authentication authentication=new UsernamePasswordAuthenticationToken(
        user, token, ②
        user.getAuthorities());
    return authentication;
}
```

① verification and signing keys are the same for symmetric algorithms

② there is no real use for the token in the authResult. It was placed in the password position in the event we wanted to locate it.

432.7. JwtUtil: parseToken() Helper Methods

The following helper method extracts the authority strings stored in the (parsed) token and wraps them in `GrantedAuthority` objects to be used by the authorization framework.

JwtUtil parseToken() Helper Methods

```
protected List<GrantedAuthority> getGrantedAuthorities(Claims claims) {  
    List<String> authorities = (List) claims.get(jwtConfig.getAuthoritiesKey());  
    return authorities==null ? Collections.emptyList() :  
        authorities.stream()  
            .map(a->new SimpleGrantedAuthority(a)) ①  
            .collect(Collectors.toList());  
}
```

① converting authority strings from token into `GrantedAuthority` objects used by Spring security framework

The following helper method returns the verify key to be the same as the signing key.

Example JwtConfig parseToken() Helper Methods

```
public class JwtConfig {  
    public SecretKey getSigningKey() {  
        return Keys.hmacShaKeyFor(getKey().getBytes(Charset.forName("UTF-8")));  
    }  
    public SecretKey getVerifyKey() {  
        return getSigningKey();  
    }  
}
```

Chapter 433. JwtAuthenticationFilter

The `JwtAuthenticationFilter` is the target filter for generating new bearer tokens. It accepts POSTS to a configured `/api/login` URI with the username and password, authenticates those credentials, generates a bearer token with JWS, and returns that value in the `Authorization` header. The following is an example of making the end-to-end authentication call. Notice the bearer token returned. We will need this value in follow-on calls.

Example End-to-End Authentication Call

```
$ curl -v -X POST http://localhost:8080/api/login -d '{"username":"frasier", "password":"password"}'  
> POST /api/login HTTP/1.1  
< HTTP/1.1 200  
< Authorization: Bearer eyJhbGciOiJIUzIwMjQ4NTk0OTgwMTAyLCJleHAiOjE40TQ50DM3MDIsImF1dGgiOlsiUFJJQ0VfQ0hFQ0siLCJST0xFX0NVU1RPTUVSI119.u2MmzTxaDoVNFGGCrAcWBusS_NS2NndZXkaT964hLgcDTvCYAW_sXtTxRw8g_13
```

The `JwtAuthenticationFilter` delegates much of the detail work handling the header and JWS token to the `JwtUtil` class shown earlier.

JwtAuthenticationFilter

```
@Slf4j  
public class JwtAuthenticationFilter extends UsernamePasswordAuthenticationFilter {  
    private final JwtUtil jwtUtil;
```

433.1. JwtAuthenticationFilter Relationships

The `JwtAuthenticationFilter` fills out the abstract workflow of the `AbstractAuthenticationProcessingFilter` by implementing two primary methods: `attemptAuthentication()` and `successfulAuthentication()`.

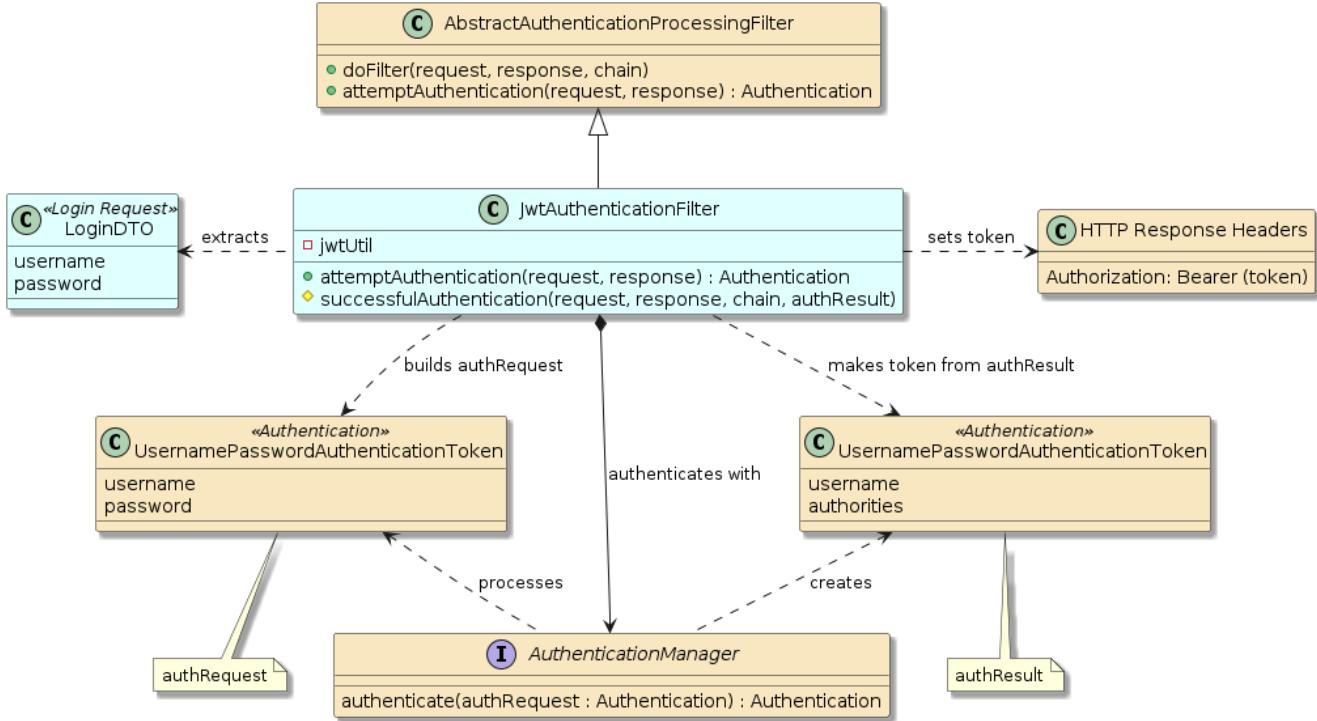


Figure 184. *JwtAuthenticationFilter Relationships*

The `attemptAuthenticate()` callback is used to perform all the steps necessary to authenticate the caller. Unsuccessful attempts are returned to the caller immediately with a 401/Unauthorized status.

The `successfulAuthentication()` callback is used to generate the JWS token from the authResult and return that in the response header. The call is returned immediately to the caller with a 200/OK status and an Authorization header containing the constructed token.

433.2. JwtAuthenticationFilter: Constructor

The filter constructor sets up the object to only listen to POSTs against the configured loginUri. The base class we are extending holds onto the `AuthenticationManager` used during the `attemptAuthentication()` callback.

JwtAuthenticationFilter Constructor

```

public JwtAuthenticationFilter(JwtConfig jwtConfig, AuthenticationManager authm) {
    super(new AntPathRequestMatcher(jwtConfig.getLoginUri(), "POST"));
    this.jwtUtil = new JwtUtil(jwtConfig);
    setAuthenticationManager(authm);
}
  
```

433.3. JwtAuthenticationFilter: attemptAuthentication()

The `attemptAuthentication()` method has two core jobs: obtain credentials and authenticate.

- The credentials could have been obtained in a number of different ways. I have simply chosen to create a DTO class with username and password to carry that information.
- The credentials are stored in an `Authentication` object that acts as the authRequest. The authResult from the `AuthenticationManager` is returned from the callback.

Any failure (`getCredentials()` or `authenticate()`) will result in an `AuthenticationException` thrown.

JwtAuthenticationFilter attemptAuthentication()

```
@Override
public Authentication attemptAuthentication(
    HttpServletRequest request, HttpServletResponse response)
    throws AuthenticationException { ①

    LoginDTO login = getCredentials(request);
    UsernamePasswordAuthenticationToken authRequest =
        new UsernamePasswordAuthenticationToken(login.getUsername(), login.
    getPassword());

    Authentication authResult = getAuthenticationManager().authenticate(authRequest);
    return authResult;
}
```

① any failure to obtain a successful `Authentication` result will throw an `AuthenticationException`

433.4. JwtAuthenticationFilter: attemptAuthentication() DTO

The `LoginDTO` is a simple POJO class that will get marshalled as JSON and placed in the body of the POST.

JwtAuthenticationFilter attemptAuthentication() DTO

```
package info.ejava.examples.svc.auth.cart.security.jwt;

import lombok.Getter;
import lombok.Setter;

@Setter
@Getter
public class LoginDTO {
    private String username;
    private String password;
}
```

433.5. JwtAuthenticationFilter: attemptAuthentication() Helper Method

We can use the Jackson Mapper to easily unmarshal the POST payload into DTO form any rethrown any failed parsing as a `BadCredentialsException`. Unfortunately for debugging, the default 401/Unauthorized response to the caller does not provide details we supply here but I guess that is a good thing when dealing with credentials and login attempts.

JwtAuthenticationFilter attemptAuthentication() Helper Method

```
...
import com.fasterxml.jackson.databind.ObjectMapper;
...
protected LoginDTO getCredentials(HttpServletRequest request) throws
AuthenticationException {
    try {
        return new ObjectMapper().readValue(request.getInputStream(), LoginDTO.class);
    } catch (IOException ex) {
        log.info("error parsing loginDTO", ex);
        throw new BadCredentialsException(ex.getMessage()); ①
    }
}
```

① `BadCredentialsException` extends `AuthenticationException`

433.6. JwtAuthenticationFilter: successfulAuthentication()

The `successfulAuthentication()` is called when authentication was successful. It has two primary jobs: encode the authenticated result in a JWS token and set the value in the response header.

JwtAuthenticationFilter successfulAuthentication()

```
@Override
protected void successfulAuthentication(
    HttpServletRequest request, HttpServletResponse response, FilterChain chain,
    Authentication authResult) throws IOException, ServletException {

    String token = jwtUtil.generateToken(authResult); ①
    log.info("generated token={}", token);
    jwtUtil.setToken(response, token); ②
}
```

① `authResult` represented within the claims of the JWS

② caller given the JWS token in the response header

This callback fully overrides the parent method to eliminate setting the `SecurityContext` and issuing a redirect. Neither have relevance in this situation. The authenticated caller will not require a

`SecurityContext` now—this is the login. The `SecurityContext` will be set as part of the call to the operation.

Chapter 434. JwtAuthorizationFilter

The `JwtAuthorizationFilter` is responsible for realizing any provided JWS bearer tokens as an authResult within the current `SecurityContext` on the way to invoking an operation. The following end-to-end operation call shows the caller supplying the bearer token in order to identify themselves to the server implementing the operation. The example operation uses the username of the current `SecurityContext` as a key to locate information for the caller.

Example Operation Call with JWS Bearer Token

```
$ curl -v -X POST http://localhost:8080/api/carts/items?name=thing \
-H "Authorization: Bearer
eyJhbGciOiJIUzIwMjQ4NCJ9.eyJzdWIiOiJmcmFzaWVyiwiawF0IjoxNTk0OTgwMTAyLCJleHAiOjE4OTQ5ODM3M
DIssImF1dGgiOlSiUFJJQ0VfQ0hFQ0sILCJST0xFX0NVU1RPTUVSIi19.u2MmzTxaDoVNFGGCnrgAcWBusS_NS2N
ndZXkaT964hLgcDTvCYAW_sXtTxRw8g_13"
> POST /api/carts/items?name=thing HTTP/1.1
...
< HTTP/1.1 200
>{"username":"frasier","items":["thing"]} ① ②
```

- ① username is encoded within the JWS token
 - ② cart with items is found by username

The `JwtAuthorizationFilter` did not seem to match any of the Spring-provided authentication filters—so I directly extended a generic filter support class that assures it will only get called once per request.

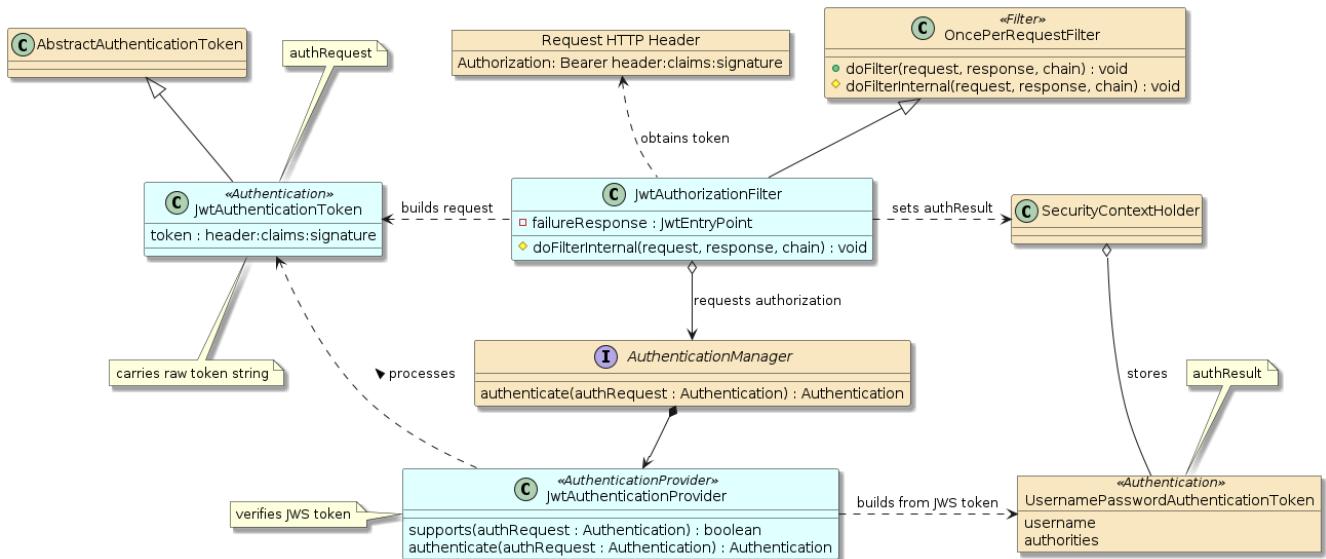
This class also relies on `JwtUtil` to implement the details of working with the JWS bearer token

JwtAuthorizationFilter

```
public class JwtAuthorizationFilter extends OncePerRequestFilter {  
    private final JwtUtil jwtUtil;  
    private final AuthenticationManager authenticationManager;  
    private final AuthenticationEntryPoint failureResponse = new JwtEntryPoint();
```

434.1. JwtAuthorizationFilter Relationships

The `JwtAuthorizationFilter` extends the generic framework of `OncePerRequestFilter` and performs all of its work in the `doFilterInternal()` callback.



The `JwtAuthorizationFilter` obtains the raw JWS token from the request header, wraps the token in the `JwsAuthenticationToken` authRequest and requests authentication from the `AuthenticationManager`. Placing this behavior in an `AuthenticationProvider` was optional but seemed to be consistent with the framework. It also provided the opportunity to lookup further user details if ever required.

Supporting the `AuthenticationManager` is the `JwtAuthenticationProvider`, which verifies the JWS token and re-builds the authResult from the JWS token claims.

The filter finishes by setting the authResult in the `SecurityContext` prior to advancing the chain further towards the operation call.

434.2. JwtAuthorizationFilter: Constructor

The `JwtAuthorizationFilter` relies on the `JwtUtil` helper class to implement the meat of the JWS token details. It also accepts an `AuthenticationManager` that is assumed to be populated with the `JwtAuthenticationProvider`.

JwtAuthorizationFilter Constructor

```

public JwtAuthorizationFilter(JwtConfig jwtConfig, AuthenticationManager
authenticationManager) {
    jwtUtil = new JwtUtil(jwtConfig);
    this.authenticationManager = authenticationManager;
}
  
```

434.3. JwtAuthorizationFilter: doFilterInternal()

Like most filters the `JwtAuthorizationFilter` initially determines if there is anything to do. If there is no `Authorization` header with a "Bearer" token, the filter is quietly bypassed and the filter chain is advanced.

If a token is found, we request authentication — where the JWS token is verified and converted

back into an `Authentication` object to store in the `SecurityContext` as the authResult.

Any failure to complete authentication when the token is present in the header will result in the chain terminating and an error status returned to the caller.

`JwtAuthorizationFilter doFilterInternal()`

```
@Override  
protected void doFilterInternal(HttpServletRequest request, HttpServletResponse  
response, FilterChain filterChain)  
    throws ServletException, IOException {  
  
    String token = jwtUtil.getToken(request);  
    if (token == null) { //continue on without JWS authn/authz  
        filterChain.doFilter(request, response); ①  
        return;  
    }  
  
    try {  
        Authentication authentication = new JwtAuthenticationToken(token); ②  
        Authentication authenticated = authenticationManager.authenticate  
(authentication);  
        SecurityContextHolder.getContext().setAuthentication(authenticated); ③  
        filterChain.doFilter(request, response); //continue chain to operation ④  
    } catch (AuthenticationException fail) {  
        failureResponse.commence(request, response, fail); ⑤  
        return; //end the chain and return error to caller  
    }  
}
```

- ① chain is quietly advanced forward if there is no token found in the request header
- ② simple authRequest wrapper for the token
- ③ store the authenticated user in the `SecurityContext`
- ④ continue the chain with the authenticated user now present in the `SecurityContext`
- ⑤ issue an error response if token is present but we are unable to complete authentication

434.4. JwtAuthenticationToken

The `JwtAuthenticationToken` has a simple job—carry the raw JWS token string through the authentication process and be able to provide it to the `JwtAuthenticationProvider`. I am not sure whether I gained much by extending the `AbstractAuthenticationToken`. The primary requirement was to implement the `Authentication` interface. As you can see, the implementation simply carries the value and returns it for just about every question asked. It will be the job of `JwtAuthenticationProvider` to turn that token into an `Authentication` instance that represents the authResult, carrying authorities and other properties that have more exposed details.

JwtAuthenticationToken Class

```
public class JwtAuthenticationToken extends AbstractAuthenticationToken {  
    private final String token;  
    public JwtAuthenticationToken(String token) {  
        super(Collections.emptyList());  
        this.token = token;  
    }  
    public String getToken() {  
        return token;  
    }  
    @Override  
    public Object getCredentials() {  
        return token;  
    }  
    @Override  
    public Object getPrincipal() {  
        return token;  
    }  
}
```

The `JwtAuthenticationProvider` class implements two key methods: `supports()` and `authenticate()`

JwtAuthenticationProvider Class

```
public class JwtAuthenticationProvider implements AuthenticationProvider {  
    private final JwtUtil jwtUtil;  
    public JwtAuthenticationProvider(JwtConfig jwtConfig) {  
        jwtUtil = new JwtUtil(jwtConfig);  
    }  
    @Override  
    public boolean supports(Class<?> authentication) {  
        return JwtAuthenticationToken.class.isAssignableFrom(authentication);  
    }  
    @Override  
    public Authentication authenticate(Authentication authentication)  
        throws AuthenticationException {  
        try {  
            String token = ((JwtAuthenticationToken)authentication).getToken();  
            Authentication authResult = jwtUtil.parseToken(token);  
            return authResult;  
        } catch (JwtException ex) {  
            throw new BadCredentialsException(ex.getMessage());  
        }  
    }  
}
```

The `supports()` method returns true only if the token type is the `JwtAuthenticationToken` type.

The `authenticate()` method obtains the raw token value, confirms its validity, and builds an

`Authentication authResult` from its claims. The result is simply returned to the `AuthenticationManager` and the calling filter.

Any error in `authenticate()` will result in an `AuthenticationException`. The most likely is an expired token—but could also be the result of a munged token string.

434.5. JwtEntryPoint

The `JwtEntryPoint` class implements an `AuthenticationEntryPoint` interface that is used elsewhere in the framework for cases when an error handler is needed because of an `AuthenticationException`. We are using it within the `JwtAuthorizationProvider` to report an error with authentication—but you will also see it show up elsewhere.

JwtEntryPoint

```
package info.ejava.examples.svc.auth.cart.security.jwt;

import org.springframework.http.HttpStatus;
import org.springframework.security.core.AuthenticationException;
import org.springframework.security.web.AuthenticationEntryPoint;

public class JwtEntryPoint implements AuthenticationEntryPoint {
    @Override
    public void commence(HttpServletRequest request, HttpServletResponse response,
                         AuthenticationException authException) throws IOException {
        response.sendError(HttpStatus.UNAUTHORIZED.value(), authException.getMessage());
    }
}
```

Chapter 435. API Security Configuration

With all the supporting framework classes in place, I will now show how we can wire this up. This, of course, takes us back to the `WebSecurityConfigurer` class.

- We inject required beans into the configuration class. The only thing that is new is the `JwtConfig @ConfigurationProperties` class. The `UserDetailsService` provides users/passwords and authorities from a database
- `configure(HttpSecurity)` is where we setup our `FilterChainProxy`
- `configure(AuthenticationManagerBuilder)` is where we setup our `AuthenticationManager` used by our filters in the `FilterChainProxy`.

API Security Configuration

```
@Configuration
@Order(0)
@RequiredArgsConstructor
@EnableConfigurationProperties(JwtConfig.class) ①
public class APIConfiguration extends WebSecurityConfigurerAdapter {
    private final JwtConfig jwtConfig; ②
    private final UserDetailsService jdbcUserDetailsService; ③

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        // details here ...
    }
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        //details here ...
    }
}
```

① enabling the `JwtConfig` as a `@ConfigurationProperties` bean

② injecting the `JwtConfig` bean into our configuration class

③ injecting a source of user details (i.e., username/password and authorities)

435.1. API Authentication Manager Builder

The `configure(AuthenticationManagerBuilder)` configures the builder with two `AuthenticationProviders`

- one containing real users/passwords and authorities
- a second with the ability to instantiate an `Authentication` from a JWS token

API Authentication Manager Builder

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
```

```

        auth.userDetailsService(jdbcUserDetailsService); ①
        auth.authenticationProvider(new JwtAuthenticationProvider(jwtConfig));
    }

```

① configuring an `AuthenticationManager` with both the `UserDetailsService` and our new `JwtAuthenticationProvider`

The `UserDetailsService` was injected because it required setup elsewhere. However, the `JwtAuthenticationProvider` is stateless—getting everything it needs from a startup configuration and the authentication calls.

435.2. API HttpSecurity Key JWS Parts

The following snippet shows the key parts to wire in the JWS handling.

- we register the `JwtAuthenticationFilter` to handle authentication of logins
- we register the `JwtAuthorizationFilter` to handle restoring the `SecurityContext` when the caller presents a valid JWS bearer token
- not required—but we register a custom error handler that leaks some details about why the caller is being rejected when receiving a 403/Forbidden

```

@Override
protected void configure(HttpSecurity http) throws Exception {
    //...
    http.addFilterAt(new JwtAuthenticationFilter(jwtConfig, ①
        authenticationManager(),
        UsernamePasswordAuthenticationFilter.class);
    http.addFilterAfter(new JwtAuthorizationFilter(jwtConfig, ②
        authenticationManager(),
        JwtAuthenticationFilter.class);
    http.exceptionHandling(cfg->cfg.defaultAuthenticationEntryPointFor( ③
        new JwtEntryPoint(),
        new AntPathRequestMatcher("/api/**")));
}

http.authorizeRequests(cfg->cfg.antMatchers("/api/login").permitAll());
http.authorizeRequests(cfg->cfg.antMatchers("/api/carts/**").authenticated());
}

```

① `JwtAuthenticationFilter` being registered at location normally used for `UsernamePasswordAuthenticationFilter`

② `JwtAuthorizationFilter` being registered after the authn filter

③ adding an optional error reporter

435.3. API HttpSecurity Full Details

The following shows the full contents of the `configure(HttpSecurity)` method. In this view you can see how FORM and BASIC Auth have been disabled and we are operating in a stateless mode with

various header/CORS options enabled.

API HttpSecurity Full Details

```
@Override  
protected void configure(HttpSecurity http) throws Exception {  
    http.requestMatchers(m->m.antMatchers("/api/**"));  
    http.httpBasic(cfg->cfg.disable());  
    http.formLogin(cfg->cfg.disable());  
    http.headers(cfg->{  
        cfg.xssProtection().disable();  
        cfg.frameOptions().disable();  
    });  
    http.csrf(cfg->cfg.disable());  
    http.cors();  
    http.sessionManagement(cfg->cfg  
        .sessionCreationPolicy(SessionCreationPolicy.STATELESS));  
  
    http.addFilterAt(new JwtAuthenticationFilter(jwtConfig,  
        authenticationManager()),  
        UsernamePasswordAuthenticationFilter.class);  
    http.addFilterAfter(new JwtAuthorizationFilter(jwtConfig,  
        authenticationManager()),  
        JwtAuthenticationFilter.class);  
    http.exceptionHandling(cfg->cfg.defaultAuthenticationEntryPointFor(  
        new JwtEntryPoint(),  
        new AntPathRequestMatcher("/api/**")));  
  
    http.authorizeRequests(cfg->cfg.antMatchers("/api/login").permitAll());  
    http.authorizeRequests(cfg->cfg.antMatchers("/api/whoami").permitAll());  
    http.authorizeRequests(cfg->cfg.antMatchers("/api/carts/**").authenticated());  
}
```

Chapter 436. Example JWT/JWS Application

Now that we have thoroughly covered the addition of the JWT/JWS to the security framework of our application, it is time to look at the application and with a focus on authorizations. I have added a few unique aspects since the previous lecture's example use of `@PreAuthorize`.

- we are using JWT/JWS—of course
- access annotations are applied to the service interface versus controller class
- access annotations inspect the values of the input parameters

436.1. Roles and Role Inheritance

I have reused the same users, passwords, and role assignments from the authorities example and will demonstrate with the following users.

- ROLE_ADMIN - `sam`
- ROLE_CLERK - `woody`
- ROLE_CUSTOMER - `norm` and `frasier`

However, role inheritance is only defined for ROLE_ADMIN inheriting all accesses from ROLE_CLERK. None of the roles inherit from ROLE_CUSTOMER.

Role Inheritance

```
@Bean
public RoleHierarchy roleHierarchy() {
    RoleHierarchyImpl roleHierarchy = new RoleHierarchyImpl();
    roleHierarchy.setHierarchy(StringUtils.join(Arrays.asList(
        "ROLE_ADMIN > ROLE_CLERK"), System.lineSeparator()));
    return roleHierarchy;
}
```

436.2. CartsService

We have a simple CartsService with a Web API and service implementation. The code below shows the interface to the service. It has been annotated with `@PreAuthorize` expressions that use the Spring Expression Language to evaluate the principal from the SecurityContext and parameters of the call.

CartsService

```
package info.ejava.examples.svc.auth.cart.services;

import info.ejava.examples.svc.auth.cart.dto.CartDTO;
import org.springframework.security.access.prepost.PreAuthorize;
```

```

public interface CartsService {

    @PreAuthorize("#username == authentication.name and hasRole('CUSTOMER')") ①
    CartDTO createCart(String username);

    @PreAuthorize("#username == authentication.name or hasRole('CLERK')") ②
    CartDTO getCart(String username);

    @PreAuthorize("#username == authentication.name") ③
    CartDTO addItem(String username, String item);

    @PreAuthorize("#username == authentication.name or hasRole('ADMIN')") ④
    boolean removeCart(String username);
}

```

- ① anyone with the **CUSTOMER** role can create a cart but it must be for their username
- ② anyone can get their own cart and anyone with the **CLERK** role can get anyone's cart
- ③ users can only add item to their own cart
- ④ users can remove their own cart and anyone with the **ADMIN** role can remove anyone's cart

436.3. Login

The following shows creation of tokens for four example users

Sam

```

$ curl -v -X POST http://localhost:8080/api/login -d '{"username":"sam",
"password":"password"}' ①
> POST /api/login HTTP/1.1
< HTTP/1.1 200
< Authorization: Bearer
eyJhbGciOiJIUzIwMjQ4NCJ9.eyJzdWIiOiJzYW0iLCJpYXQiOjE1OTUwMTcwNDQsImV4cCI6MTg5NTAyMDY0NCwiY
XV0acI6WyJST0xFX0FETUlOI119.ICzAn1r2UyrgpGJQSYk9uqxMAAq9QC1Dw7GKe0NiGvCyTasMfWSStrqxV6U
it-cb4

```

- ① **sam** has role **ADMIN** and inherits role **CLERK**

Woody

```

$ curl -v -X POST http://localhost:8080/api/login -d '{"username":"woody",
"password":"password"}' ①
> POST /api/login HTTP/1.1
< HTTP/1.1 200
< Authorization: Bearer
eyJhbGciOiJIUzIwMjQ4NCJ9.eyJzdWIiOiJ3b29keSIsImhlhdCI6MTU5NTAxNzA1MSwiZXhwIjoxODk1MDIwNjUxL
CJhdXRoIjpbIlJPTEVfQ0xFUksiXX0.kreSFPgTIr2heGMLcjHFrlydvhPZKR7Iy4F6b76WNIvAkbZVhfymbQ
xekuPL-Ai

```

① woody has role CLERK

Norm and Frasier

```
$ curl -v -X POST http://localhost:8080/api/login -d '{"username":"norm", "password":"password"}' ①
> POST /api/login HTTP/1.1
< HTTP/1.1 200
< Authorization: Bearer
eyJhbGciOiJIUzM4NCJ9.eyJzdWIiOiJub3JtIiwiaWF0IjoxNTk1MDE3MDY1LCJleHAiOjE4OTUwMjA2NjUsImF1dGgiOlslUk9MRV9DVVNUT01FUijdfQ.UX4yPDu0LzWdEA0bbJli0tZ7ePU1RSIH_o_hayPrlmNxhjU5DL6XQ42iRCLLuFgw

$ curl -v -X POST http://localhost:8080/api/login -d '{"username":"frasier", "password":"password"}' ①
> POST /api/login HTTP/1.1
< HTTP/1.1 200
< Authorization: Bearer
eyJhbGciOiJIUzM4NCJ9.eyJzdWIiOiJmcmFzaWVyIiwiaWF0IjoxNTk1MDE3MDcxLCJleHAiOjE4OTUwMjA2NzEsImF1dGgiOlslUFJJQ0hFQ0siLCJST0xFX0NVU1RPTUVSI19.ELAe5foIL_u2QyhpjwDoqQbL4Hl1Ikuir9CJPdOT80w2lI5Z1GQY6ZaKvW883txI
```

① norm and frasier have role CUSTOMER

436.4. createCart()

The access rules for `createCart()` require the caller be a customer and be creating a cart for their username.

`createCart()` Access Rules

```
@PreAuthorize("#username == authentication.name and hasRole('CUSTOMER')") ①
CartDTO createCart(String username); ①
```

① `#username` refers to the `username` method parameter

Woody is unable to create a cart because he lacks the `CUSTOMER` role.

Woody Unable to Create Cart

```
$ curl -X GET http://localhost:8080/api/whoAmI -H "Authorization: Bearer
eyJhbGciOiJIUzM4NCJ9.eyJzdWIiOiJ3b29keSIsImhdCI6MTU5NTAxNzA1MSwiZXhwIjoxODk1MDIwNjUxLCJhdXRoIjpbIlJPTEVfQ0xFUksiXX0.kreSFPgTIR2heGMLcjHFrglydvhPZKR7Iy4F6b76WNIvAkbZVhfymbQxekuPL-Ai" #woody
[woody, [ROLE_CLERK]]
```



```
$ curl -X POST http://localhost:8080/api/carts -H "Authorization: Bearer
eyJhbGciOiJIUzM4NCJ9.eyJzdWIiOiJ3b29keSIsImhdCI6MTU5NTAxNzA1MSwiZXhwIjoxODk1MDIwNjUxLCJhdXRoIjpbIlJPTEVfQ0xFUksiXX0.kreSFPgTIR2heGMLcjHFrglydvhPZKR7Iy4F6b76WNIvAkbZVhfymbQxekuPL-Ai" #woody
```

```
{"url": "http://localhost:8080/api/carts", "message": "Forbidden", "description": "caller[woody] is forbidden from making this request", "timestamp": "2020-07-17T20:24:14.159507Z"}
```

Norm is able to create a cart because he has the **CUSTOMER** role.

Norm Can Create Cart

```
$ curl -X GET http://localhost:8080/api/whoAmI -H "Authorization: Bearer eyJhbGciOiJIUzM4NCJ9eyJzdWIiOiJub3JtIiwiaWF0IjoxNTk1MDE3MDY1LCJleHAiOjE4OTUwMjA2NjUsImF1dGgiOlslsiUk9MRV9DVVNUT01FUijdfQ.UX4yPDu0LzWdEA0bbJli0tZ7ePU1RSIH_o_hayPrlmNxhjU5DL6XQ42iRCLLuFgw" #norm  
[norm, [ROLE_CUSTOMER]]  
  
$ curl -X POST http://localhost:8080/api/carts -H "Authorization: Bearer eyJhbGciOiJIUzM4NCJ9eyJzdWIiOiJub3JtIiwiaWF0IjoxNTk1MDE3MDY1LCJleHAiOjE4OTUwMjA2NjUsImF1dGgiOlslsiUk9MRV9DVVNUT01FUijdfQ.UX4yPDu0LzWdEA0bbJli0tZ7ePU1RSIH_o_hayPrlmNxhjU5DL6XQ42iRCLLuFgw" #norm  
{"username": "norm", "items": []}
```

436.5. **addItem()**

The **addItem()** access rules only allow users to add items to their own cart.

addItem() Access Rules

```
@PreAuthorize("#username == authentication.name")  
CartDTO addItem(String username, String item);
```

Frasier is forbidden from adding items to Norm's cart because his identity does not match the username for the cart.

Frasier Cannot Add to Norms Cart

```
$ curl -X GET http://localhost:8080/api/whoAmI -H "Authorization: Bearer eyJhbGciOiJIUzM4NCJ9eyJzdWIiOiJmcmFzaWVyIiwiaWF0IjoxNTk1MDE3MDcxLCJleHAiOjE4OTUwMjA2NzEsImF1dGgiOlslsiUFJJQ0VfQ0hFQ0siLCJST0xFX0NVU1RPTUVSI119.ELAe5foIL_u2QyhpjwDoqQbL4Hl1Ikuir9CJPdOT80w2lI5Z1GQY6ZaKvW883txI" #frasier  
[frasier, [PRICE_CHECK, ROLE_CUSTOMER]]  
  
$ curl -X POST "http://localhost:8080/api/carts/items?username=norm&name=chardonnay" -H "Authorization: Bearer eyJhbGciOiJIUzM4NCJ9eyJzdWIiOiJmcmFzaWVyIiwiaWF0IjoxNTk1MDE3MDcxLCJleHAiOjE4OTUwMjA2NzEsImF1dGgiOlslsiUFJJQ0VfQ0hFQ0siLCJST0xFX0NVU1RPTUVSI119.ELAe5foIL_u2QyhpjwDoqQbL4Hl1Ikuir9CJPdOT80w2lI5Z1GQY6ZaKvW883txI" #frasier  
{ "url": "http://localhost:8080/api/carts/items?username=norm&name=chardonnay", "message": "Forbidden", "description": "caller[frasier] is forbidden from making this request", "timestamp": "2020-07-17T20:40:10.451578Z" } ①
```

① **frasier** received a 403/Forbidden error when attempting to add to someone else's cart

Norm can add items to his own cart because his username matches the username of the cart.

Norm Can Add to His Own Cart

```
$ curl -X POST http://localhost:8080/api/carts/items?name=beer -H "Authorization: Bearer eyJhbGciOiJIUzIwMjQ4NCJ9.eyJzdWIiOiJub3JtIiwiaWF0IjoxNTk1MDE3MDY1LCJleHAiOjE4OTUwMjA2NjUsImF1dGgiOlslsiUk9MRV9DVVNUT01FUijdfQ.UX4yPDu0LzWdEA0bbJli0tZ7ePU1RSIH_o_hayPrlmNxhjU5DL6XQ42iRCLLuFgw" #norm {"username": "norm", "items": ["beer"]}
```

436.6. `getCart()`

The `getCart()` access rules only allow users to get their own cart, but also allows users with the **CLERK** role to get anyone's cart.

getCart() Access Rules

```
@PreAuthorize("#username == authentication.name or hasRole('CLERK')") ②
CartDTO getCart(String username);
```

Frasier cannot get Norm's cart because anyone lacking the **CLERK** role can only get a cart that matches their authenticated username.

Frasier Cannot Get Norms Cart

```
$ curl -X GET http://localhost:8080/api/carts?username=norm -H "Authorization: Bearer eyJhbGciOiJIUzIwMjQ4NCJ9.eyJzdWIiOiJmcmFzaWVyIiwiaWF0IjoxNTk1MDE3MDcxLCJleHAiOjE4OTUwMjA2NzEsImF1dGgiOlslsiUFJJQ0VfQ0hFQ0s1LCJST0xFX0NVU1RPTUVSIl19.ELAe5foIL_u2QyhpjwDoqQbL4Hl1Ikuir9CJPdOT80w2lI5Z1GQY6ZaKvW883txI" #frasier
>{"url": "http://localhost:8080/api/carts?username=norm", "message": "Forbidden", "description": "caller[frasier] is forbidden from making this request", "timestamp": "2020-07-17T20:44:05.899192Z"}
```

Norm can get his own cart because the username of the cart matches the authenticated username of his accessing the cart.

Norm Can Get Norms Cart

```
$ curl -X GET http://localhost:8080/api/carts -H "Authorization: Bearer eyJhbGciOiJIUzIwMjQ4NCJ9.eyJzdWIiOiJub3JtIiwiaWF0IjoxNTk1MDE3MDY1LCJleHAiOjE4OTUwMjA2NjUsImF1dGgiOlslsiUk9MRV9DVVNUT01FUijdfQ.UX4yPDu0LzWdEA0bbJli0tZ7ePU1RSIH_o_hayPrlmNxhjU5DL6XQ42iRCLLuFgw" #norm {"username": "norm", "items": ["beer"]}
```

Woody can get Norm's cart because he has the **CLERK** role.

Woody Can Get Norms Cart

```
$ curl -X GET http://localhost:8080/api/carts?username=norm -H "Authorization: Bearer eyJhbGciOiJIUzM4NCJ9.eyJzdWIiOiJ3b29keSIsImhdCI6MTU5NTAxNzA1MSwiZXhwIjoxODk1MDIwNjUxLCJhdXRoiJpbILJPTEVfQ0xFUksiXX0.kreSFPgTIR2heGMLcjHFrglydvhPZKR7Iy4F6b76WNIvAkbZVhfymbQxekuPL-Ai" #woody
{"username":"norm","items":["beer"]}
```

436.7. removeCart()

The `removeCart()` access rules only allow carts to be removed by their owner or by someone with the **ADMIN** role.

removeCart() Access Rules

```
@PreAuthorize("#username == authentication.name or hasRole('ADMIN')")
boolean removeCart(String username);
```

Woody cannot remove Norm's cart because his authenticated username does not match the cart and he lacks the **ADMIN** role.

Woody Cannot Remove Norms Cart

```
$ curl -X DELETE http://localhost:8080/api/carts?username=norm -H "Authorization: Bearer eyJhbGciOiJIUzM4NCJ9.eyJzdWIiOiJ3b29keSIsImhdCI6MTU5NTAxNzA1MSwiZXhwIjoxODk1MDIwNjUxLCJhdXRoiJpbILJPTEVfQ0xFUksiXX0.kreSFPgTIR2heGMLcjHFrglydvhPZKR7Iy4F6b76WNIvAkbZVhfymbQxekuPL-Ai" #woody
{"url":"http://localhost:8080/api/carts?username=norm","message":"Forbidden","description":"caller[woody] is forbidden from making this request","timestamp":"2020-07-17T20:48:40.866193Z"}
```

Sam can remove Norm's cart because he has the **ADMIN** role. Once Sam deletes the cart, Norm receives a 404/Not Found because it is no longer there.

Sam Can Remove Norms Cart

```
$ curl -X GET http://localhost:8080/api/whoAmI -H "Authorization: Bearer eyJhbGciOiJIUzM4NCJ9.eyJzdWIiOiJzYW0iLCJpYXQiOjE1OTUwMTcwNDQsImV4cCI6MTg5NTAyMDY0NCwiYXV0aCI6WyJST0xFX0FETUlOI19.ICzAn1r2UyrgPJQSYk9uqxMAAq9QC1Dw7GKe0NiGvCyTasMfWSStrqxV6Uiit-cb4" #sam
[sam, [ROLE_ADMIN]]
```



```
$ curl -X DELETE http://localhost:8080/api/carts?username=norm -H "Authorization: Bearer eyJhbGciOiJIUzM4NCJ9.eyJzdWIiOiJzYW0iLCJpYXQiOjE1OTUwMTcwNDQsImV4cCI6MTg5NTAyMDY0NCwiYXV0aCI6WyJST0xFX0FETUlOI19.ICzAn1r2UyrgPJQSYk9uqxMAAq9QC1Dw7GKe0NiGvCyTasMfWSStrqxV6Uiit-cb4" #sam
```

```
$ curl -X GET http://localhost:8080/api/carts -H "Authorization: Bearer eyJhbGciOiJIUzIwMjY1NCJ9.eyJzdWIiOiJub3JtIiwiaWF0IjoxNTk1MDE3MDY1LCJleHAiOjE4OTUwMjA2NjUsImF1dGgiOlSiUk9MRV9DVVNUT01FUijdfQ.UX4yPDu0LzWdEA0bbJli0tZ7ePU1RSIH_o_hayPrlmNxhjU5DL6XQ42iRCLLuFgw" #norm
{"url":"http://localhost:8080/api/carts","message":"Not Found","description":"no cart found for norm","timestamp":"2020-07-17T20:50:59.465210Z"}
```

Chapter 437. Summary

I don't know about you — but I had fun with that!

To summarize — In this module, we learned:

- to separate the authentication from the operation call such that the operation call could be in a separate server or even an entirely different service
- what is a JSON Web Token (JWT) and JSON Web Secret (JWS)
- how trust is verified using JWS
- how to write and/or integrate custom authentication and authorization framework classes to implement an alternate security mechanism in Spring/Spring Boot
- how to leverage Spring Expression Language to evaluate parameters and properties of the `SecurityContext`

Unit Integration Testing

copyright © 2024 jim stafford (jim.stafford@jhu.edu)

Chapter 438. Introduction

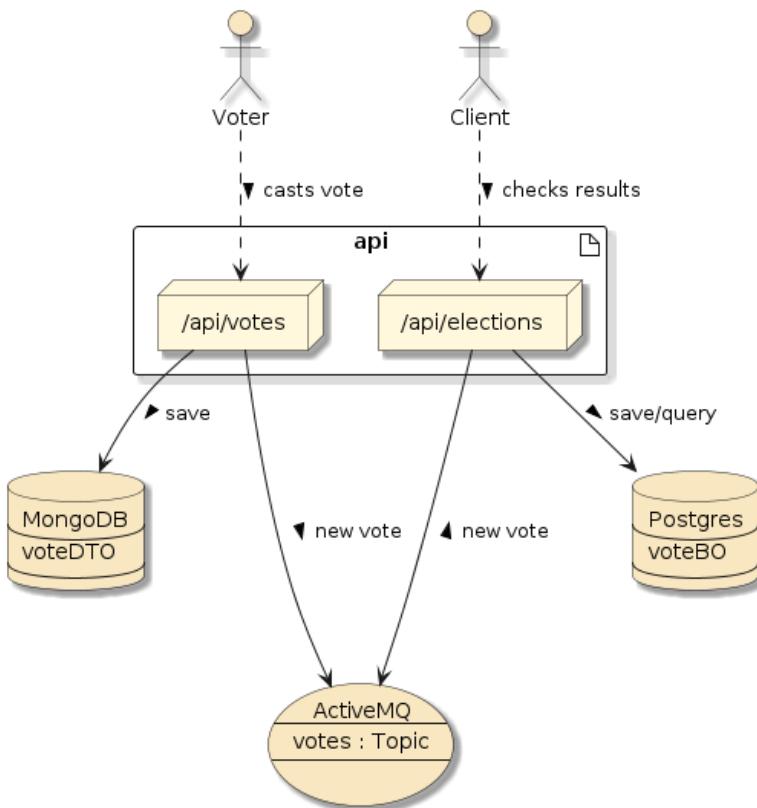
In the testing lectures I made a specific point to separate the testing concepts of

- focusing on a single class with stubs and mocks
- integrating multiple classes through a Spring context
- having to manage separate processes using the Maven integration test phases and plugins

Having only a single class under test meets most definitions of "**unit testing**". Having to manage multiple processes satisfies most definitions of "**integration testing**". Having to integrate multiple classes within a single JVM using a single JUnit test is a bit of a middle ground because it takes less heroics (thanks to modern test frameworks) and can be moderately fast.

I have termed the middle ground "**unit integration testing**" in an earlier lecture and labeled them with the suffix "NTest" to signify that they should run within the surefire unit test Maven phase and will take more time than a mocked unit test. In this lecture, I am going to expand the scope of "unit integration test" to include simulated resources like databases and JMS servers. This will allow us to write tests that are moderately efficient but more fully test layers of classes and their underlying resources within the context of a thread that is more representative of an end-to-end usecase.

Given an application like the following with databases and a JMS server...



- how can we test application interaction with a real instance of the database?
- how can we test the integration between two processes communicating with JMS events?
- how can we test timing aspects between disparate user and application events?
- how can we test a **measured** amount of end-to-end tests with the scope of our unit integration tests?

Figure 185. Votes Application

438.1. Goals

You will learn:

- how to integrate MongoDB into a Spring Boot application
- how to integrate a Relational Database into a Spring Boot application
- how to integrate a JMS server into a Spring Boot application
- how to implement an unit integration test using embedded resources

438.2. Objectives

At the conclusion of this lecture and related exercises, you will be able to:

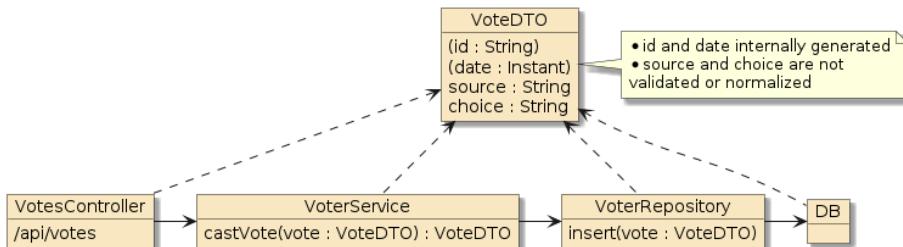
1. embed a simulated MongoDB within a JUnit test using Flapdoodle
2. embed an in-memory JMS server within a JUnit test using ActiveMQ
3. embed a relational database within a JUnit test using H2
4. verify an end-to-end test case using a unit integration test

Chapter 439. Votes and Elections Service

For this example, I have created two moderately parallel services—Votes and Elections—that follow a straight forward controller, service, repository, and database layering.

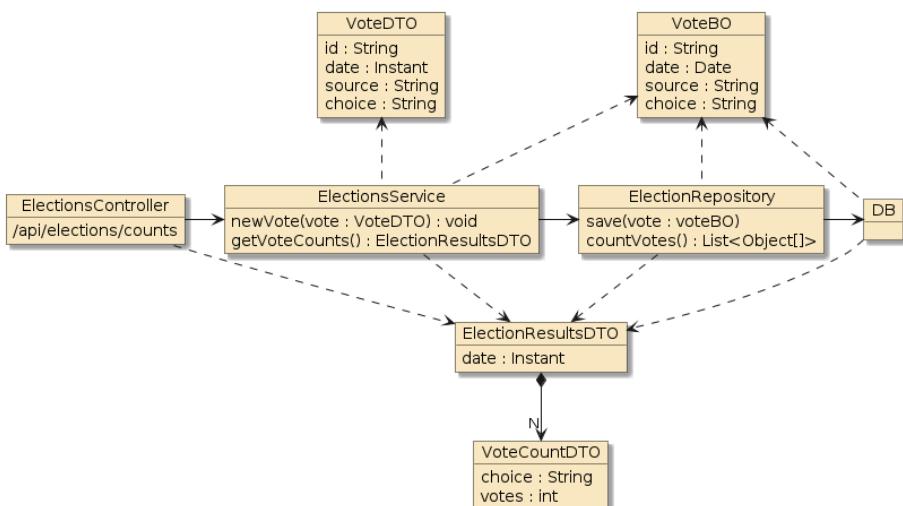
439.1. Main Application Flows

Table 30. Service Controller/Database Dependencies



The Votes Service accepts a vote (VoteDTO) from a caller and stores that directly in a database (MongoDB).

Figure 186. VotesService



The Elections service transforms received votes (VoteDTO) into database entity instances (VoteBO) and stores them in a separate database (Postgres) using Java Persistence API (JPA). The service uses that persisted information to provide election results from aggregated queries of the database.

Figure 187. ElectionsService

The fact that the applications use MongoDB, Postgres Relational DB, and JPA will only be a very small part of the lecture material. However, it will serve as a basic template of how to integrate these resources for much more complicated unit integration tests and deployment scenarios.

439.2. Service Event Integration

The two services are integrated through a set of Aspects, ApplicationEvent, and JMS logic that allow the two services to be decoupled from one another.

Table 31. Async Dependencies

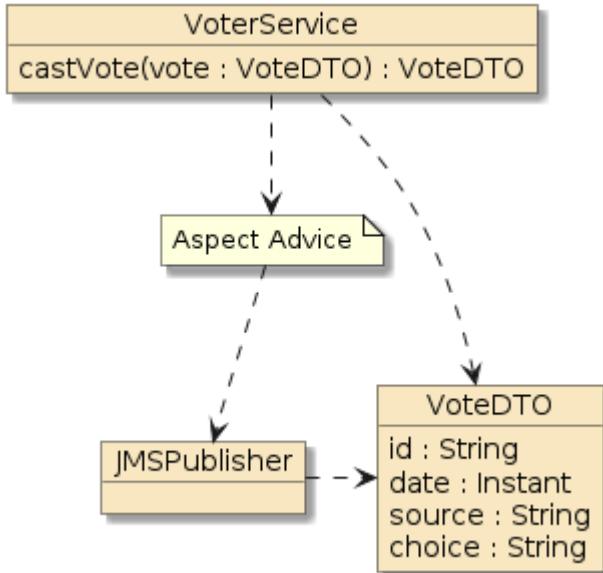


Figure 188. Votes Service

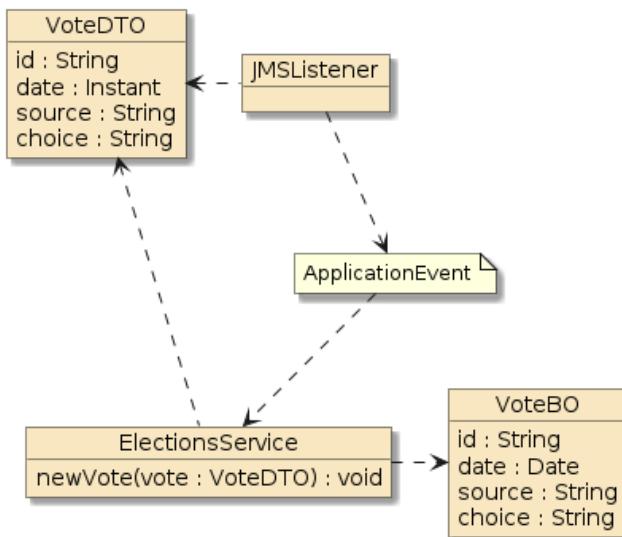


Figure 189. Elections Service

The fact that the applications use JMS will only be a small part of the lecture material. However, it too will serve as a basic template of how to integrate another very pertinent resource for distributed systems.

The Votes Service events layer defines a pointcut on the successful return of the `VotesService.castVote()` and publishes the resulting vote (VoteDTO)—with newly assigned ID and date—to a JMS destination.

Chapter 440. Physical Architecture

I described five (5) functional services in the previous section: Votes, Elections, MongoDB, Postgres, and ActiveMQ (for JMS).

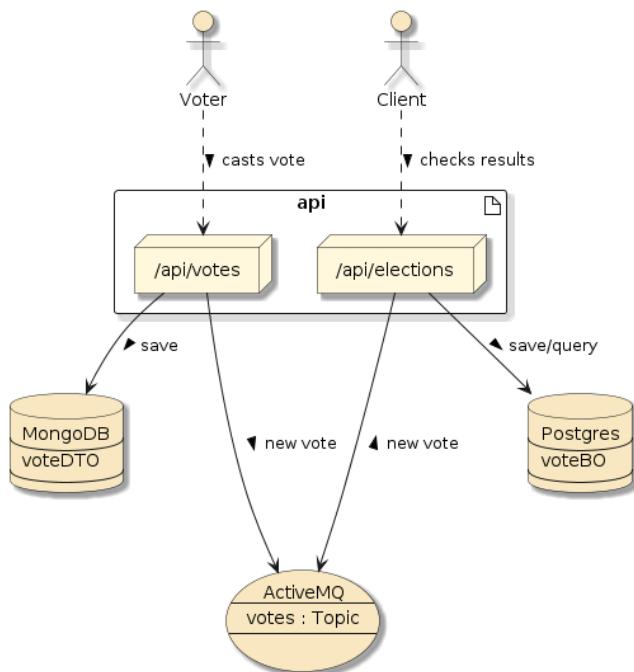


Figure 190. Physical Architecture

440.1. Unit Integration Test Physical Architecture

For unit integration tests, we will use a single JUnit JVM with the Spring Boot Services and the three resources embedded using the following options:

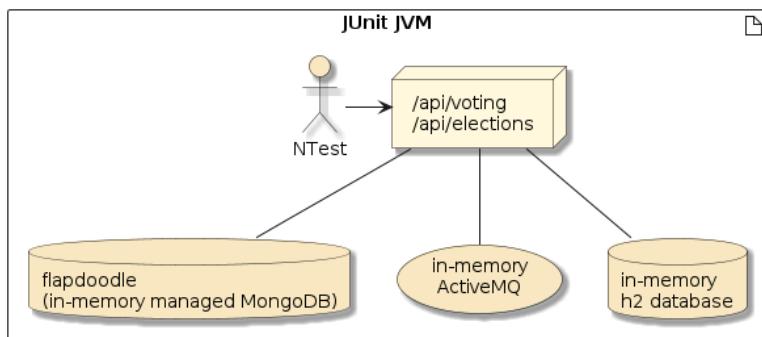


Figure 191. Unit Integration Testing Physical Architecture

- [Flapdoodle](#) - an open source initiative that markets itself to implementing an embedded MongoDB. It is incorrect to call the entirety of Flapdoodle "embedded". The management of MongoDB is "embedded" but a real server image is being downloaded and executed behind the scenes.

- another choice is [fongo](#). Neither are truly embedded and neither had much activity in the past 2 years, but flapdoodle has twice as many stars on github and has been active more recently (as of Aug 2020).

- [H2 Database](#) in memory RDBMS we used for user management during the later security topics
- [ActiveMQ \(Classic\)](#) used in embedded mode

Chapter 441. Mongo Integration

In this section we will go through the steps of adding the necessary MongoDB dependencies to implement a MongoDB repository and simulate that with an in-memory DB during unit integration testing.

441.1. MongoDB Maven Dependencies

As with most starting points with Spring Boot—we can bootstrap our application to implement a MongoDB repository by forming an dependency on [spring-boot-starter-data-mongodb](#).

Primary MongoDB Maven Dependency

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb</artifactId>
</dependency>
```

That brings in a few driver dependencies that will also activate the [MongoAutoConfiguration](#) to establish a default [MongoClient](#) from properties.

MongoDB Starter Dependencies

```
[INFO] +- org.springframework.boot:spring-boot-starter-data-mongodb:jar:3.1.2:compile
[INFO] |   +- org.mongodb:mongodb-driver-sync:jar:4.9.1:compile
[INFO] |   |   +- org.mongodb:bson:jar:4.9.1:compile
[INFO] |   |   \- org.mongodb:mongodb-driver-core:jar:4.9.1:compile
[INFO] |   |       \- org.mongodb:bson-record-codec:jar:4.9.1:runtime
[INFO] |   \- org.springframework.data:spring-data-mongodb:jar:4.1.2:compile
```

441.2. Test MongoDB Maven Dependency

For testing, we add a dependency on [de.flapdoodle.embed.mongo](#). By setting scope to [test](#), we avoid deploying that with our application outside of our module testing.

Test MongoDB Maven Dependency

```
<dependency>
    <groupId>de.flapdoodle.embed</groupId>
    <artifactId>de.flapdoodle.embed.mongo.spring30x</artifactId>
    <scope>test</scope>
</dependency>
```

The [flapdoodle](#) dependency brings in the following artifacts.

Flapdoodle Dependencies

```
[INFO] +- de.flapdoodle.embed:de.flapdoodle.embed.mongo.spring30x:jar:4.5.2:test
[INFO] |  +- de.flapdoodle.embed:de.flapdoodle.embed.mongo:jar:4.5.1:test
[INFO] |  |  +- de.flapdoodle.embed:de.flapdoodle.embed.process:jar:4.5.0:test
[INFO] |  |  |  +- de.flapdoodle.reverse:de.flapdoodle.reverse:jar:1.5.2:test
[INFO] |  |  |  |  +- de.flapdoodle.graph:de.flapdoodle.graph:jar:1.2.3:test
[INFO] |  |  |  |  |  \- org.jgrapht:jgrapht-core:jar:1.4.0:test
[INFO] |  |  |  |  |  \- org.jheaps:jheaps:jar:0.11:test
[INFO] |  |  |  |  |  \- de.flapdoodle.java8:de.flapdoodle.java8:jar:1.3.2:test
[INFO] |  |  |  |  +- org.apache.commons:commons-compress:jar:1.22:test
[INFO] |  |  |  |  +- net.java.dev.jna:jna:jar:5.13.0:test
[INFO] |  |  |  |  +- net.java.dev.jna:jna-platform:jar:5.13.0:test
[INFO] |  |  |  |  \- de.flapdoodle:de.flapdoodle.os:jar:1.2.7:test
[INFO] |  |  \-
[INFO] |  de.flapdoodle.embed:de.flapdoodle.embed.mongo.packageresolver:jar:4.4.1:test
...
...
```

441.3. MongoDB Properties

The following lists a core set of MongoDB properties we will use no matter whether we are in test or production. If we implement the most common scenario of a single single database—things get pretty easy to work through properties. Otherwise we would have to provide our own [MongoClient @Bean](#) factories to target specific instances.

Core MongoDB Properties

```
#mongo
spring.data.mongodb.authentication-database=admin ①
spring.data.mongodb.database=votes_db ②
```

① identifies the mongo database with user credentials

② identifies the mongo database for our document collections

441.4. MongoDB Repository

Spring Data provides a very nice repository layer that can handle basic CRUD and query capabilities with a simple interface definition that extends [MongoRepository<T, ID>](#). The following shows an example declaration for a [VoteDTO](#) POJO class that uses a String for a primary key value.

MongoDB VoterRepository Declaration

```
import info.ejava.examples.svc.docker.votes.dto.VoteDTO;
import org.springframework.data.mongodb.repository.MongoRepository;

public interface VoterRepository extends MongoRepository<VoteDTO, String> {
```

441.5. VoteDTO MongoDB Document Class

The following shows the MongoDB document class that doubles as a Data Transfer Object (DTO) in the controller and JMS messages.

Example VoteDTO MongoDB Document Class

```
import lombok.*;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;
import java.time.Instant;

@Document("votes") ①
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class VoteDTO {
    @Id
    private String id; ②
    private Instant date;
    private String source;
    private String choice;
}
```

① MongoDB `Document` class mapped to the `votes` collection

② `VoteDTO.id` property mapped to `_id` field of MongoDB collection

Example Stored VoteDTO Document

```
{
    "_id": {"$oid": "5f3204056ac44446600b57ff"},
    "date": {"$date": {"$numberLong": "1597113349837"}},
    "source": "jim",
    "choice": "quisp",
    "_class": "info.ejava.examples.svc.docker.votes.dto.VoteDTO"
}
```

441.6. Sample MongoDB/VoterRepository Calls

The following snippet shows the injection of the repository into the service class and two sample calls. At this point in time, it is only important to notice that our simple repository definition gives us the ability to insert and count documents (and more!!!).

Sample MongoDB/VoterRepository Calls

```
@Service
@RequiredArgsConstructor ①
public class VoterServiceImpl implements VoterService {
```

```
private final VoterRepository voterRepository; ①

@Override
public VoteDTO castVote(VoteDTO newVote) {
    newVote.setId(null);
    newVote.setDate(Instant.now());
    return voterRepository.insert(newVote); ②
}

@Override
public long getTotalVotes() {
    return voterRepository.count(); ③
}
```

① using constructor injection to initialize service with repository

② repository inherits ability to insert new documents

③ repository inherits ability to get count of documents

This service is then injected into the controller and accessed through the [/api/votes](#) URI. At this point we are ready to start looking at the details of how to report the new votes to the [ElectionsService](#).

Chapter 442. ActiveMQ Integration

In this section we will go through the steps of adding the necessary ActiveMQ dependencies to implement a JMS publish/subscribe and simulate that with an in-memory JMS server during unit integration testing.

442.1. ActiveMQ Maven Dependencies

The following lists the dependencies we need to implement the Aspects and JMS capability within the application. Artemis is an offshoot of ActiveMQ and the [spring-boot-starter-artemis](#) dependency provides the jakarta.jms support we need within Spring Boot 3.

ActiveMQ Primary Maven Dependency

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-artemis</artifactId>
</dependency>

<!-- dependency adds a runtime server to allow running with embedded topic -->
<dependency>
    <groupId>org.apache.activemq</groupId>
    <artifactId>artemis-jakarta-server</artifactId>
</dependency>
```

The Artemis starter brings in the following dependencies and activates the [ActiveMQAutoConfiguration](#) class that will setup a JMS connection based on properties.

Artemis Starter Dependencies

```
[INFO] +- org.springframework.boot:spring-boot-starter-artemis:jar:3.1.2:compile
[INFO] |   +- org.springframework:spring-jms:jar:6.0.11:compile
[INFO] |   |   +- org.springframework:spring-messaging:jar:6.0.11:compile
[INFO] |   |   \- org.springframework:spring-tx:jar:6.0.11:compile
[INFO] |   \- org.apache.activemq:artemis-jakarta-client:jar:2.28.0:compile
[INFO] |       \- org.apache.activemq:artemis-selector:jar:2.28.0:compile
```

If we enable connection pooling, we need the following [pooled-jms](#) dependency.

JMS Connection Pooling Dependency

```
<!-- jmsTemplate connection polling -->
<dependency>
    <groupId>org.messaginghub</groupId>
```

```
<artifactId>pooled-jms</artifactId>
</dependency>
```

442.2. ActiveMQ Unit Integration Test Properties

The following lists the core property required by ActiveMQ in all environments. Without the `pub-sub-domain` property defined, ActiveMQ defaults to a queue model—which will not allow our integration tests to observe the (topic) traffic flow we want. Without connection pools, each new JMS interaction will result in a physical open/close of the connection to the server. Enabling connection pools allows a pool of physical connections to be physically open and then shared across multiple JMS interactions without physically closing each time.

ActiveMQ Core Properties

```
#activemq
#to have injected beans use JMS Topics over Queues -- by default
spring.jms.pub-sub-domain=true ①

#requires org.messaginghub:pooled-jms dependency
#https://activemq.apache.org/spring-support
spring.artemis.pool.enabled=true ②
spring.artemis.pool.max-connections=5
```

① tells ActiveMQ/Artemis to use topics versus queues

② enables JMS clients to share a pool of connections so that each JMS interaction does not result in a physical open/close of the connection

The following lists the properties that are unique to the local unit integration tests.

ActiveMQ Test Properties

```
#activemq
spring.artemis.broker-url=tcp://activemq:61616 ①
```

① activemq will establish in-memory destinations

442.3. Service Joinpoint Advice

I used Aspects to keep the Votes Service flow clean of external integration and performed that by enabling Aspects using the `@EnableAspectJAutoProxy` annotation and defining the following `@Aspect` class, joinpoint, and advice.

Example Service Joinpoint Advice

```
@Aspect
@Component
@RequiredArgsConstructor
public class VoterAspects {
```

```

private final VoterJMS votePublisher;

@Pointcut("within(info.ejava.examples.svc.docker.votes.services.VoterService+)")
public void voterService(){} ①

@Pointcut("execution(*..VoteDTO castVote(..))")
public void castVote(){} ②

@AfterReturning(value = "voterService() && castVote()", returning = "vote")
public void afterVoteCast(VoteDTO vote) { ③
    try {
        votePublisher.publish(vote);
    } catch (IOException ex) {
        ...
    }
}

```

① matches all calls implementing the `VoterService` interface

② matches all calls called `castVote` that return a `VoteDTO`

③ injects returned `VoteDTO` from matching calls and calls publish to report event

442.4. JMS Publish

The publishing of the new vote event using JMS is done within the `VoterJMS` class using an injected `jmsTemplate` and `ObjectMapper`. Essentially, the method marshals the `VoteDTO` object into a JSON text string and publishes that in a `TextMessage` to the "votes" topic.

JMS Publisher Code

```

@Component
@RequiredArgsConstructor
public class VoterJMS {
    private final JmsTemplate jmsTemplate; ①
    private final ObjectMapper jsonMapper; ②
    ...

    public void publish(VoteDTO vote) throws JsonProcessingException {
        final String json = jsonMapper.writeValueAsString(vote); ③

        jmsTemplate.send("votes", new MessageCreator() { ④
            @Override
            public Message createMessage(Session session) throws JMSException {
                return session.createTextMessage(json); ⑤
            }
        });
    }
}

```

- ① inject a jmsTemplate supplied by ActiveMQ starter dependency
- ② inject ObjectMapper that will marshal objects to JSON
- ③ marshal vote to JSON string
- ④ publish the JMS message to the "votes" topic
- ⑤ publish vote JSON string using a JMS `TextMessage`

442.5. ObjectMapper

The `ObjectMapper` that was injected in the `VoterJMS` class was built using a custom factory that configured it to use formatting and write timestamps in ISO format versus binary values.

ObjectMapper Factory

```
@Bean
@Order(Ordered.LOWEST_PRECEDENCE) //allows to be overridden using property
public Jackson2ObjectMapperBuilderCustomizer jacksonCustomizer() {
    return builder -> builder.indentOutput(true)
        .featuresToDisable(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS);
}

@Bean
public ObjectMapper jsonMapper(Jackson2ObjectMapperBuilder builder) {
    return builder.createXmlMapper(false).build();
}
```

442.6. JMS Receive

The JMS receive capability is performed within the same `VoterJMS` class to keep JMS implementation encapsulated. The class implements a method accepting a JMS `TextMessage` annotated with `@JmsListener`. At this point we could have directly called the `ElectionsService` but I chose to go another level of indirection and simply issue an `ApplicationEvent`.

JMS Receive Code

```
@Component
@RequiredArgsConstructor
public class VoterJMS {
    ...
    private final ApplicationEventPublisher eventPublisher;
    private final ObjectMapper jsonMapper;

    @JmsListener(destination = "votes") ②
    public void receive(TextMessage message) throws JMSException { ①
        String json = message.getText();
        try {
            VoteDTO vote = jsonMapper.readValue(json, VoteDTO.class); ③
            eventPublisher.publishEvent(new NewVoteEvent(vote)); ④
        }
    }
}
```

```

        } catch (JsonProcessingException ex) {
            //...
        }
    }
}

```

- ① implements a method receiving a JMS `TextMessage`
- ② method annotated with `@JmsListener` against the `votes` topic
- ③ JSON string unmarshalled into a `VoteDTO` instance
- ④ Simple `NewVote` POJO event created and issued internal

442.7. EventListener

An `EventListener @Component` is supplied to listen for the application event and relay that to the `ElectionsService`.

Example Application Event Listener

```

import org.springframework.context.event.EventListener;

@Component
@RequiredArgsConstructor
public class ElectionListener {
    private final ElectionsService electionService;

    @EventListener ②
    public void newVote(NewVoteEvent newVoteEvent) { ①
        electionService.addVote(newVoteEvent.getVote()); ③
    }
}

```

- ① method accepts `NewVoteEvent` POJO
- ② method annotated with `@EventListener` looking for application events
- ③ method invokes `addVote` of `ElectionsService` when `NewVoteEvent` occurs

At this point we are ready to look at some of the implementation details of the Elections Service.

Chapter 443. JPA Integration

In this section we will go through the steps of adding the necessary dependencies to implement a JPA repository and simulate that with an in-memory RDBMS during unit integration testing.

443.1. JPA Core Maven Dependencies

The Elections Service uses a relational database and interfaces with that using Spring Data and Java Persistence API (JPA). To do that, we need the following core dependencies defined. The starter sets up the default JDBC DataSource and JPA layer. The `postgresql` dependency provides a client for Postgres and one that takes responsibility for Postgres-formatted JDBC URLs.

RDBMS Core Dependencies

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <scope>runtime</scope>
</dependency>
```

There are too many (~20) dependencies to list that come in from the `spring-boot-starter-data-jpa` dependency. You can run `mvn dependency:tree` yourself to look, but basically it brings in Hibernate and connection pooling. The supporting libraries for Hibernate and JPA are quite substantial.

443.2. JPA Test Dependencies

During unit integration testing we add the H2 database dependency to provide another option.

JPA Test Dependencies

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>test</scope>
</dependency>
```

443.3. JPA Properties

The test properties include a direct reference to the in-memory H2 JDBC URL. I will explain the use of Flyway next, but this is considered optional for this case because Spring Data will trigger auto-schema population for in-memory databases.

```
#rdbms
spring.datasource.url=jdbc:h2:mem:users ①
spring.jpa.show-sql=true ②

# optional: in-memory DB will automatically get schema generated
spring.flyway.enabled=true ③
```

① JDBC in-memory H2 URL

② show SQL so we can see what is occurring between service and database

③ optionally turn on Flyway migrations

443.4. Database Schema Migration

Unlike the NoSQL MongoDB, relational databases have a strict schema that defines how data is stored. That must be accounted for in all environments. However — the way we do it can vary:

- Auto-Generation - the simplest way to configure a development environment is to use JPA/Hibernate auto-generation. This will delegate the job of populating the schema to Hibernate at startup. This is perfect for dynamic development stages where schema is changing constantly. This is unacceptable for production and other environments where we cannot lose all of our data when we restart our application.
- Manual Schema Manipulation - relational database schema can get more complex than what can get auto-generated and even auto-generated schema normally passes through the review of human eyes before making it to production. Deployment can be a manually intensive and likely the choice of many production environments where database admins must review, approve, and possibly execute the changes.

Once our schema stabilizes, we can capture the changes to a versioned file and use the Flyway plugin to automate the population of schema. If we do this during unit integration testing, we get a chance to supply a more tested product for production deployment.

443.5. Flyway RDBMS Schema Migration

Flyway is a schema migration library that can do forward (free) and reverse (at a cost) RDBMS schema migrations. We include Flyway by adding the following dependency to the application.

Flyway Maven Dependency

```
<dependency>
  <groupId>org.flywaydb</groupId>
  <artifactId>flyway-core</artifactId>
  <scope>runtime</scope>
</dependency>
```

The Flyway test properties include the JDBC URL that we are using for the application and a flag to

enable.

Flyway Test Properties

```
spring.datasource.url=jdbc:h2:mem:users ①  
spring.flyway.enabled=true
```

① Flyway makes use of the Spring Boot database URL

443.6. Flyway RDBMS Schema Migration Files

We feed the Flyway plugin schema migrations that move the database from version N to version N+1, etc. The default directory for the migrations is in `db/migration` of the classpath. The directory is populated with files that are executed in order according to a name syntax that defaults to `V#_#_#_description` (double underscore between last digit of version and first character of description; the number of digits in the version is not mandatory)

Flyway Migration File Structure

```
dockercompose-votes-svc/src/main/resources/  
`-- db  
    '-- migration  
        |-- V1.0.0__initial_schema.sql  
        '-- V1.0.1__expanding_choice_column.sql
```

The following is an example of a starting schema (V1_0_0).

Create Table/Index Example Migration #1

```
create table vote (  
    id varchar(50) not null,  
    choice varchar(40),  
    date timestamp,  
    source varchar(40),  
    constraint vote_pkey primary key(id)  
);  
  
comment on table vote is 'countable votes for election';
```

The following is an example of a follow-on migration after it was determined that the original `choice` column size was too small.

Expand Table Column Size Example Migration #2

```
alter table vote alter column choice type varchar(60);
```

443.7. Flyway RDBMS Schema Migration Output

The following is an example Flyway migration occurring during startup.

Example Flyway Schema Migration Output

```
Database: jdbc:h2:mem:users (H2 2.1)
Schema history table "PUBLIC"."flyway_schema_history" does not exist yet
Successfully validated 2 migrations (execution time 00:00.016s)
Creating Schema History table "PUBLIC"."flyway_schema_history" ...
Current version of schema "PUBLIC": << Empty Schema >>
Migrating schema "PUBLIC" to version "1.0.0 - initial schema"
Migrating schema "PUBLIC" to version "1.0.1 - expanding choice column"
Successfully applied 2 migrations to schema "PUBLIC", now at version v1.0.1 (execution
time 00:00.037s)
```

For our unit integration test—we end up at the same place as auto-generation, except we are taking the opportunity to dry-run and regression test the schema migrations prior to them reaching production.

443.8. JPA Repository

The following shows an example of our JPA/ElectionRepository. Similar to the MongoDB repository—this extension will provide us with many core CRUD and query methods. However, the one aggregate query targeted for this database cannot be automatically supplied without some help. We must provide the SQL query to return the choice, vote count, and latest vote data for that choice.

JPA/ElectionRepository

```
...
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.repository.Query;

public interface ElectionRepository extends JpaRepository<VoteBO, String> {
    @Query("select choice, count(id), max(date) from VoteBO group by choice order by
count(id) DESC") ①
    public List<Object[]> countVotes(); ②
}
```

① JPA query language to return choices aggregated with vote count and latest vote for each choice

② a list of arrays—one per result row—with raw DB types is returned to caller

443.9. Example VoteBO Entity Class

The following shows the example JPA Entity class used by the repository and service. This is a standard JPA definition that defines a table override, primary key, and mapping aspects for each property in the class.

Example VoteBO Entity Class

```
...
import jakarta.persistence.*;

@Entity ①
@Table(name="VOTE") ②
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class VoteBO {
    @Id ③
    @Column(length = 50) ④
    private String id;
    @Temporal(TemporalType.TIMESTAMP)
    private Date date;
    @Column(length = 40)
    private String source;
    @Column(length = 40)
    private String choice;
}
```

① `@Entity` annotation required by JPA

② overriding default table name (`VOTEBO`)

③ JPA requires valid Entity classes to have primary key marked by `@Id`

④ column size specifications only used when generating schema—otherwise depends on migration to match

443.10. Sample JPA/ElectionRepository Calls

The following is an example service class that is injected with the `ElectionRepository` and is able to make a few sample calls. `save()` is pretty straight forward but notice that `countVotes()` requires some extra processing. The repository method returns a list of `Object[]` values populated with raw values from the database—representing choice, voteCount, and lastDate. The newest `lastDate` is used as the date of the election results. The other two values are stored within a `VoteCountDTO` object within `ElectionResultsDTO`.

Elections Service Class

```
@Service
@RequiredArgsConstructor
public class ElectionsServiceImpl implements ElectionsService {
    private final ElectionRepository votesRepository;

    @Override
    @Transactional(value = Transactional.TxType.REQUIRED)
    public void addVote(VoteDTO voteDTO) {
```

```
    VoteBO vote = map(voteDTO);
    votesRepository.save(vote); ①
}

@Override
public ElectionResultsDTO getVoteCounts() {
    List<Object[]> counts = votesRepository.countVotes(); ②

    ElectionResultsDTO electionResults = new ElectionResultsDTO();
    // ...
    return electionResults;
}
```

① `save()` inserts a new row into the database

② `countVotes()` returns a list of `Object[]` with raw values from the DB

Chapter 44. Unit Integration Test

Stepping outside of the application and looking at the actual unit integration test—we see the majority of the magical meat in the first several lines.

- `@SpringBootTest` is used to define an application context that includes our complete application plus a test configuration that is used to inject necessary test objects that could be configured differently for certain types of tests (e.g., security filter)
- The port number is randomly generated and injected into the constructor to form baseUrls. We will look at a different technique in the Testcontainers lecture that allows for more first-class support for late-binding properties.

Example Unit Integration Test

```
@SpringBootTest( classes = {ClientTestConfiguration.class, VotesExampleApp.class},
    webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT, ①
    properties = "test=true") ②
@ActiveProfiles("test") ③
@DisplayName("votes unit integration test")
public class VotesTemplateNTest {
    @Autowired ④
    private RestTemplate restTemplate;
    private final URI baseVotesUrl;
    private final URI baseElectionsUrl;

    public VotesTemplateNTest(@LocalServerPort int port) { ①
        baseVotesUrl = UriComponentsBuilder.fromUriString("http://localhost") ⑤
            .port(port)
            .path("/api/votes")
            .build().toUri();
        baseElectionsUrl = UriComponentsBuilder.fromUriString("http://localhost")
            .port(port)
            .path("/api/elections")
            .build().toUri();
    }
    ...
}
```

① configuring a local web environment with the random port# injected into constructor

② adding a `test=true` property that can be used to turn off conditional logic during tests

③ activating the `test` profile and its associated `application-test.properties`

④ `restTemplate` injected for cases where we may need authentication or other filters added

⑤ constructor forming reusable baseUrls with supplied random port value

444.1. ClientTestConfiguration

The following shows how the `restTemplate` was formed. In this case—it is extremely simple. However, as you have seen in other cases, we could have required some authentication and logging

filters to the instance and this is the best place to do that when required.

ClientTestConfiguration

```
@SpringBootConfiguration()
@EnableAutoConfiguration      //needed to setup logging
public class ClientTestConfiguration {
    @Bean
    public RestTemplate anonymousUser(RestTemplateBuilder builder) {
        RestTemplate restTemplate = builder.build();
        return restTemplate;
    }
}
```

444.2. Example Test

The following shows a very basic example of an end-to-end test of the Votes Service. We use the baseUrl to cast a vote and then verify that it was accurately recorded.

Example test

```
@Test
public void cast_vote() {
    //given - a vote to cast
    Instant before = Instant.now();
    URI url = baseVotesUrl;
    VoteDTO voteCast = create_vote("voter1", "quip");
    RequestEntity<VoteDTO> request = RequestEntity.post(url).body(voteCast);

    //when - vote is casted
    ResponseEntity<VoteDTO> response = restTemplate.exchange(request, VoteDTO.class);

    //then - vote is created
    then(response.getStatusCode()).isEqualTo(HttpStatus.CREATED);
    VoteDTO recordedVote = response.getBody();
    then(recordedVote.getId()).isNotEmpty();
    then(recordedVote.getDate()).isAfterOrEqual(before);
    then(recordedVote.getSource()).isEqualTo(voteCast.getSource());
    then(recordedVote.getChoice()).isEqualTo(voteCast.getChoice());
}
```

At this point in the lecture we have completed covering the important aspects of forming an unit integration test with embedded resources in order to implement end-to-end testing on a small scale.

Chapter 445. Summary

At this point we should have a good handle on how to add external resources (e.g., MongoDB, Postgres, ActiveMQ) to our application and configure our integration unit tests to operate end-to-end using either simulated or in-memory options for the real resource. This gives us the ability to identify more issues early before we go into more manually intensive integration or production. In this following lectures—I will be expanding on this topic to take on several Docker-based approaches to integration testing.

In this module, we learned:

- how to integrate MongoDB into a Spring Boot application
 - and how to unit integration test MongoDB code using Flapdoodle
- how to integrate a ActiveMQ server into a Spring Boot application
 - and how to unit integration test JMS code using an embedded ActiveMQ server
- how to integrate a Postgres into a Spring Boot application
 - and how to unit integration test relational code using an in-memory H2 database
- how to implement an unit integration test using embedded resources

Unresolved directive in jhu784-notes.adoc - include::/builds/ejava-javaee/ejava-springboot-docs/courses/jhu784-notes/target/resources/docs/asciidoc/testcontainers-votesnotes.adoc[leveloffset=0]

Unresolved directive in jhu784-notes.adoc - include::/builds/ejava-javaee/ejava-springboot-docs/courses/jhu784-notes/target/resources/docs/asciidoc/testcontainers-spock-votesnotes.adoc[leveloffset=0]