

Enterprise Java (Spring/Spring Boot) Course Notes

jim stafford

Fall 2021 v2021-08-29: Built: 2021-12-08 07:50 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	3
2.4. Student Background	4
2.5. Student Commitment	4
2.6. Course Text(s)	4
2.7. Required Software	5
2.8. Course Structure	5
2.9. Grading	6
2.10. Grading Policy	6
2.11. Academic Integrity	7
2.12. Instructor Availability	7
2.13. Communication Policy	7
2.14. Office Hours	7
3. Course Assignments	8
3.1. General Requirements	8
3.2. Submission Guidelines	8
4. Syllabus	10
Development Environment Setup	13
5. Introduction	14
5.1. Goals	14
5.2. Objectives	14
6. Software Setup	15
6.1. Java JDK (immediately)	15
6.2. Maven 3 (immediately)	15
6.3. Git Client (immediately)	16
6.4. Java IDE (immediately)	18
6.5. Web API Client tool (not immediately)	18
6.6. Optionally Install Docker (not immediately)	18
6.7. MongoDB (later)	20
6.8. Heroku Account (later)	20
6.9. Mongo Atlas Account (later)	20
Introduction to Enterprise Java Frameworks	21
7. Introduction	22
7.1. Goals	22

7.2. Objectives.....	22
8. Code Reuse.....	23
8.1. Code Reuse Trade-offs.....	23
8.2. Code Reuse Constructs.....	23
8.3. Code Reuse Styles.....	23
9. Frameworks	25
9.1. Framework Informal Description.....	25
9.2. Framework Characteristics	25
10. Framework Enablers	27
10.1. Dependency Injection.....	27
10.2. POJO	27
10.3. Component	27
10.4. Bean.....	27
10.5. Container	28
10.6. Interpose	28
11. Language Impact on Frameworks.....	31
11.1. XML Configurations.....	31
11.2. Annotations	31
11.3. Lambdas	32
12. Key Frameworks	33
12.1. CGI Scripts	33
12.2. JavaEE	33
12.3. Spring	33
12.4. Jakarta Persistence API (JPA).....	34
12.5. Spring Data	34
12.6. Spring Boot	34
13. Summary.....	36
Pure Java Main Application.....	37
14. Introduction	38
14.1. Goals	38
14.2. Objectives	38
15. Simple Java Class with a Main	39
16. Project Source Tree	40
17. Building the Java Archive (JAR) with Maven	41
17.1. Add Core pom.xml Document.....	41
17.2. Add Optional Elements to pom.xml	41
17.3. Define Plugin Versions	42
17.4. pluginManagement vs. plugins.....	43
18. Build the Module	44
19. Project Build Tree	46
20. Resulting Java Archive (JAR).....	47

21. Execute the Application	48
22. Configure Application as an Executable JAR	49
22.1. Add Main-Class property to MANIFEST.MF	49
22.2. Automate Additions to MANIFEST.MF using Maven	49
23. Execute the JAR versus just adding to classpath	50
24. Configure pom.xml to Test	51
24.1. Execute JAR as part of the build	52
25. Summary	53
Simple Spring Boot Application	54
26. Introduction	55
26.1. Goals	55
26.2. Objectives	55
27. Spring Boot Maven Dependencies	56
28. Parent POM	57
28.1. Define Version for Spring Boot artifacts	57
28.2. Import <code>springboot-dependencies-plugin</code>	57
29. Local Child/Leaf Module POM	59
29.1. Declare pom inheritance in the child pom.xml	59
29.2. Declare dependency on artifacts used	59
30. Simple Spring Boot Applicaton Java Class	60
30.1. Module Source Tree	60
30.2. <code>@SpringBootApplication</code> Aggregate Annotation	61
31. Spring Boot Executable JAR	62
31.1. Building the Spring Boot Executable JAR	62
31.2. Java MANIFEST.MF properties	63
31.3. JAR size	64
31.4. JAR Contents	64
31.5. Execute Command Line	65
32. Add a Component to Output Message and Args	67
32.1. <code>@Component</code> Annotation	67
32.2. Interface: <code>CommandLineRunner</code>	67
32.3. <code>@ComponentScan</code> Tree	68
33. Running the Spring Boot Application	69
33.1. Implementation Note	69
34. Directory Structure	70
35. Summary	71
Race Registration/Results Assignment 0	72
36. Part A: Build Pure Java Application JAR	73
36.1. Purpose	73
36.2. Overview	73
36.3. Requirements	74

36.4. Grading	74
36.5. Additional Details	75
37. Part B: Build Spring Boot Executable JAR	76
37.1. Purpose	76
37.2. Overview	76
37.3. Requirements	77
37.4. Grading	78
37.5. Additional Details	78
Bean Factory and Dependency Injection	79
38. Introduction	80
38.1. Goals	80
38.2. Objectives	80
39. Hello Service	81
39.1. Hello Service API	81
39.2. Hello Service StdOut	81
39.3. Hello Service API pom.xml	82
39.4. Hello Service Std pom.xml	82
39.5. Hello Service Interface	82
39.6. Hello Service Sample Implementation	83
39.7. Hello Service Modules Complete	84
39.8. Hello Service API Maven Build	84
39.9. Hello Service StdOut Maven Build	86
40. Application Module	88
40.1. Application Maven Dependency	88
40.2. Viewing Dependencies	89
40.3. Application Java Dependency	89
41. Dependency Injection	91
42. Spring Dependency Injection	92
42.1. @Autowired Annotation	92
42.2. Dependency Injection Flow	93
43. Bean Missing	94
43.1. Bean Missing Error Solution(s)	94
44. @Configuration classes	95
45. @Bean Factory Method	96
46. @Bean Factory Used	97
47. Factory Alternative: XML Configuration	98
48. Summary	100
Value Injection	101
49. Introduction	102
49.1. Goals	102
49.2. Objectives	102

50. @Value Annotation	103
50.1. Value Not Found	103
50.2. Value Property Provided by Command Line	104
50.3. Default Value	104
51. Constructor Injection	105
52. Property Types	106
52.1. non-String Property Types	106
52.2. Collection Property Types	106
52.3. Custom Delimiters (using Spring EL)	107
52.4. Map Property Types	108
52.5. Map Element	108
52.6. System Properties	108
52.7. Property Conversion Errors	109
53. Summary	110
Property Source	111
54. Introduction	112
54.1. Goals	112
54.2. Objectives	112
55. Property File Source(s)	113
55.1. Property File Source Example	113
55.2. Example Property File Contents	114
55.3. Alternate File Examples	115
55.4. Series of files	116
56. @PropertySource Annotation	117
57. Profiles	119
57.1. Default Profile	120
57.2. Specific Active Profile	121
57.3. Multiple Active Profiles	121
57.4. No Associated Profile	122
58. Property Placeholders	123
58.1. Placeholder Demonstration	123
58.2. Placeholder Property Files	124
58.3. Placeholder Value Defined Internally	124
58.4. Placeholder Value Defined in Profile	124
58.5. Multiple Active Profiles	125
58.6. Mixing Names, Profiles, and Location	125
59. Summary	127
Configuration Properties	128
60. Introduction	129
60.1. Goals	129
60.2. Objectives	129

61. Mapping properties to @ConfigurationProperties class	130
61.1. Mapped Java Class	130
61.2. Injection Point	131
61.3. Initial Error	131
61.4. Registering the @ConfigurationProperties class	132
61.5. Result	134
62. Metadata	135
62.1. Spring Configuration Metadata	135
62.2. Spring Configuration Processor	135
62.3. Javadoc Supported	136
62.4. Rebuild Module	136
62.5. IDE Property Help	137
63. Constructor Binding	138
63.1. Property Names Bound to Constructor Parameter Names	138
63.2. Constructor Parameter Name Mismatch	139
64. Validation	140
64.1. Validation Annotations	140
64.2. Validation Error	141
65. Boilerplate JavaBean Methods	143
65.1. Generating Boilerplate Methods with Lombok	143
65.2. Visible Generated Constructs	144
65.3. Lombok Build Dependency	145
65.4. Example Output	145
66. Relaxed Binding	148
66.1. Relaxed Binding Example JavaBean	148
66.2. Relaxed Binding Example Properties	148
66.3. Relaxed Binding Example Output	149
67. Nested Properties	150
67.1. Nested Properties JavaBean Mapping	150
67.2. Nested Properties Host JavaBean Mapping	150
67.3. Nested Properties Output	151
68. Property Arrays	152
68.1. Property Arrays Definition	152
68.2. Property Arrays Output	153
69. System Properties	154
69.1. System Properties Usage	154
70. @ConfigurationProperties Class Reuse	156
70.1. @ConfigurationProperties Class Reuse Mapping	156
70.2. @ConfigurationProperties @Bean Factory	157
70.3. Injecting ownerProps	157
70.4. Injection Matching	158

70.5. Ambiguous Injection	158
70.6. Injection @Qualifier	159
70.7. way1: Create Custom @Qualifier Annotation	159
70.8. way2: @Bean Factory Method Name as Qualifier	160
70.9. way3: Match @Bean Factory Method Name	161
70.10. Ambiguous Injection Summary	161
71. Summary	162
Auto Configuration	163
72. Introduction	164
72.1. Goals	164
72.2. Objectives	164
73. Review: Configuration Class	166
73.1. Separate @Configuration Class	166
74. Conditional Configuration	167
74.1. Property Value Condition Satisfied	167
74.2. Property Value Condition Not Satisfied	168
75. Two Primary Configuration Phases	169
76. Auto-Configuration	170
76.1. Supporting @ConfigurationProperties	170
76.2. Locating Auto Configuration Classes	171
76.3. META-INF/spring.factories Metadata File	171
76.4. Example Starter Module Source Tree	172
76.5. Example Starter Module pom.xml	172
76.6. Example Starter Implementation Dependencies	173
76.7. Application Starter Dependency	174
76.8. Starter Brings in Pertinent Dependencies	174
77. Configured Application	176
77.1. Review: Unconditional Auto-Configuration Class	176
77.2. Review: Starter Module Default	176
77.3. Produced Default Starter Greeting	177
77.4. User-Application Supplies Property Details	177
78. Auto-Configuration Conflict	178
78.1. Review: Conditional @Bean Factory	178
78.2. Potential Conflict	178
78.3. @ConditionalOnMissingBean	179
78.4. Bean Conditional Example Output	179
79. Resource Conditional and Ordering	181
79.1. Registering Second Auto-Configuration Class	181
79.2. Resource Conditional Example Output	181
80. @Primary	183
80.1. @Primary Example Output	184

81. Class Conditions	185
81.1. Class Conditional Example	185
82. Excluding Auto Configurations	186
83. Debugging Auto Configurations	187
83.1. Conditions Evaluation Report	187
83.2. Conditions Evaluation Report Example	187
83.3. Condition Evaluation Report Results	188
83.4. Actuator Conditions	188
83.5. Activating Actuator Conditions	188
83.6. Actuator Environment	189
83.7. Actuator Links	190
83.8. Actuator Environment Report	190
83.9. Actuator Specific Property Source	191
83.10. More Actuator	191
84. Summary	192
Logging	193
85. Introduction	194
85.1. Why log?	194
85.2. Why use a Logger over System.out?	194
85.3. Goals	194
85.4. Objectives	195
86. Starting References	196
87. Logging Dependencies	197
87.1. Logging Libraries	197
87.2. Spring and Spring Boot Internal Logging	198
88. Getting Started	199
88.1. System.out	199
88.2. System.out Output	199
88.3. Turning Off Spring Boot Logging	199
88.4. Getting a Logger	200
88.5. Java Util Logger Interface Example	200
88.6. JUL Example Output	201
88.7. SLF4J Logger Interface Example	202
88.8. SLF4J Example Output	202
88.9. Lombok SLF4J Declaration Example	203
88.10. Lombok Example Output	203
88.11. Lombok Dependency	204
89. Logging Levels	205
89.1. Common Level Use	205
89.2. Log Level Adjustments	206
89.3. Logging Level Example Calls	206

89.4. Logging Level Output: INFO	207
89.5. Logging Level Output: DEBUG	207
89.6. Logging Level Output: TRACE	208
89.7. Logging Level Output: WARN	208
89.8. Logging Level Output: OFF	209
90. Discarded Message Expense	210
90.1. Blind String Concatenation	210
90.2. Verbosity Check	210
90.3. SLF4J Parameterized Logging	211
90.4. Simple Performance Results: Disabled	212
90.5. Simple Performance Results: Enabled	213
91. Exception Logging	214
91.1. Exception Example Output	214
91.2. Exception Logging and Formatting	215
92. Logging Pattern	217
92.1. Default Console Pattern	217
92.2. Default Console Pattern Output	218
92.3. Variable Substitution	219
92.4. Conditional Variable Substitution	219
92.5. Date Format Pattern	220
92.6. Log Level Pattern	220
92.7. Conversion Pattern Specifiers	221
92.8. Format Modifier Impact Example	221
92.9. Example Override	222
92.10. Expensive Conversion Words	222
92.11. Example Override Output	223
92.12. Layout Fields	223
93. Loggers	224
93.1. Logger Tree	224
93.2. Logger Inheritance	224
93.3. Logger Threshold Level Inheritance	225
93.4. Logger Effective Threshold Level Inheritance	225
93.5. Example Logger Threshold Level Properties	225
93.6. Example Logger Threshold Level Output	226
94. Appenders	227
94.1. Logger has N Appenders	227
94.2. Logger Configuration Files	227
94.3. Logback Root Configuration Element	228
94.4. Retain Spring Boot Defaults	228
94.5. Appender Configuration	228
94.6. Appenders Attached to Loggers	229

94.7. Appender Tree Inheritance	230
94.8. Appender Additivity Result	230
94.9. Logger Inheritance Tree Output	231
95. Mapped Diagnostic Context	233
95.1. MDC Example	233
95.2. MDC Example Pattern	234
95.3. MDC Example Output	234
96. Markers	236
96.1. Marker Class	236
96.2. Marker Example	236
96.3. Marker Appender Filter Example	237
96.4. Marker Example Result	238
97. File Logging	239
97.1. root Logger Appenders	239
97.2. FILE Appender Output	239
97.3. Spring Boot FILE Appender Definition	240
97.4. RollingFileAppender	241
97.5. SizeAndTimeBasedRollingPolicy	241
97.6. FILE Appender Properties	242
97.7. logging.file.path	242
97.8. logging.file.name	243
97.9. logging.file.max-size Trigger	243
97.10. logging.pattern.rolling-file-name	243
97.11. Timestamp Rollover Example	244
97.12. History Compression Example	245
97.13. logging.file.max-history Example	245
97.14. logging.file.total-size-cap Index Example	246
97.15. logging.file.total-size-cap no Index Example	247
98. Custom Configurations	248
98.1. Logback Configuration Customization	248
98.2. Provided Logback Includes	248
98.3. Customization Example: Turn off Console Logging	248
98.4. LOG_FILE Property Definition	249
98.5. Customization Example: Leverage Restored Defaults	250
98.6. Customization Example: Provide Override	250
99. Spring Profiles	251
100. Summary	252
Testing	253
101. Introduction	254
101.1. Why Do We Test?	254
101.2. What are Test Levels?	254

101.3. What are some Approaches to Testing?	254
101.4. Goals	254
101.5. Objectives	255
102. Test Constructs	256
102.1. Automated Test Terminology	256
102.2. Maven Test Types	257
102.3. Test Naming Conventions	257
102.4. Lecture Test Naming Conventions	257
103. Spring Boot Starter Test Frameworks	258
103.1. Spring Boot Starter Transitive Dependencies	258
103.2. Transitive Dependency Test Tools	259
104. JUnit Background	261
104.1. JUnit 5 Evolution	262
104.2. JUnit 5 Areas	262
104.3. JUnit 5 Module JARs	263
105. Syntax Basics	264
106. JUnit Vintage Basics	265
106.1. JUnit Vintage Example Lifecycle Methods	265
106.2. JUnit Vintage Example Test Methods	266
106.3. JUnit Vintage Basic Syntax Example Output	266
107. JUnit Jupiter Basics	268
107.1. JUnit Jupiter Example Lifecycle Methods	268
107.2. JUnit Jupiter Example Test Methods	269
107.3. JUnit Jupiter Basic Syntax Example Output	269
108. Assertion Basics	271
108.1. Assertion Libraries	271
108.2. Example Library Assertions	273
108.3. Assertion Failures	274
108.4. Testing Multiple Assertions	275
108.5. Asserting Exceptions	276
108.6. Asserting Dates	278
109. Mockito Basics	280
109.1. Test Doubles	280
109.2. Mock Support	280
109.3. Mockito Example Declarations	280
109.4. Mockito Example Test	281
110. BDD Acceptance Test Terminology	283
110.1. Alternate BDD Syntax Support	283
110.2. Example BDD Syntax Support	283
110.3. Example BDD Syntax Output	285
111. Tipping Example	286

112. Review: Unit Test Basics	287
112.1. Review: POJO Unit Test Setup	287
112.2. Review: POJO Unit Test	287
112.3. Review: Mocked Unit Test Setup	288
112.4. Review: Mocked Unit Test	288
112.5. Alternative Mocked Unit Test	289
113. Spring Boot Unit Integration Test Basics	290
113.1. Adding Spring Boot to Testing	290
113.2. @SpringBootTest	290
113.3. Default @SpringBootConfiguration Class	291
113.4. Conditional Components	291
113.5. Explicit Reference to @SpringBootConfiguration	291
113.6. Explicit Reference to Components	292
113.7. Active Profiles	292
113.8. Example @SpringBootTest Unit Integration Test	293
113.9. Example @SpringBootTest Unit Integration Test Output	293
113.10. Alternative Test Slices	294
114. Mocking Spring Boot Unit Integration Tests	296
114.1. Example @SpringBoot/Mockito Test	296
115. Maven Unit Testing Basics	298
115.1. Maven Surefire Plugin	298
115.2. Filtering Tests	299
115.3. Filtering Tests Executed	299
115.4. Maven Failsafe Plugin	300
115.5. Failsafe Overhead	301
116. Summary	302
Race Registration/Results Assignment 1	303
117. Assignment 1a: App Config	305
117.1. @Bean Factory Configuration	305
117.2. Property Source Configuration	307
117.3. Configuration Properties	312
117.4. Auto-Configuration	316
118. Assignment 1b: Logging	324
118.1. Application Logging	324
118.2. Logging Efficiency	327
118.3. Appenders and Custom Log Patterns	328
119. Assignment 1c: Testing	333
119.1. Demo	333
119.2. Unit Testing	333
119.3. Mocks	336
119.4. Mocked Unit Integration Test	338

119.5. Unmocked/BDD Unit Integration Testing	340
HTTP API	342
120. Introduction	343
120.1. Goals	343
120.2. Objectives	343
121. World Wide Web (WWW)	344
121.1. Example WWW Information System	344
122. REST	345
122.1. HATEOAS	345
122.2. Clients Dynamically Discover State	345
122.3. Static Interface Contracts	346
122.4. Internet Scale	346
122.5. How RESTful?	346
122.6. Buzzword Association	346
122.7. REST-like or HTTP-based	347
122.8. Richardson MaturityModel (RMM)	347
122.9. "REST-like"/"HTTP-based" APIs	348
122.10. Uncommon REST Features Adopted	348
123. RMM Level 2 APIs	349
124. HTTP Protocol Embraced	350
125. Resource	351
125.1. Nested Resources	351
126. Uniform Resource Identifiers (URIs)	352
126.1. Related URI Terms	352
126.2. URI Generic Syntax	353
126.3. URI Component Examples	353
126.4. URI Characters and Delimiters	354
126.5. URI Percent Encoding	354
126.6. URI Case Sensitivity	355
126.7. URI Reference	355
126.8. URI Reference Terms	355
126.9. URI Naming Conventions	356
126.10. URI Variables	357
127. Methods	358
127.1. Additional HTTP Methods	358
128. Method Safety	359
128.1. Safe and Unsafe Methods	359
128.2. Violating Method Safety	359
129. Idempotent	361
129.1. Idempotent and non-Idempotent Methods	361
130. Response Status Codes	362

130.1. Common Response Status Codes	362
131. Representations	363
131.1. Content Type Headers	363
132. Links	365
133. Summary	366
Spring MVC	367
134. Introduction	368
134.1. Goals	368
134.2. Objectives	368
135. Spring Web APIs	369
135.1. Lecture/Course Focus	369
135.2. Spring MVC	369
135.3. Spring WebFlux	370
135.4. Synchronous vs Asynchronous	370
135.5. Mixing Approaches	371
135.6. Choosing Approaches	372
136. Maven Dependencies	373
137. Sample Application	374
138. Annotated Controllers	375
138.1. Class Mappings	375
138.2. Method Request Mappings	376
138.3. Default Method Response Mappings	377
138.4. Executing Sample Endpoint	377
139. RestTemplate Client	379
139.1. JUnit Integration Test Setup	379
139.2. Form Endpoint URL	379
139.3. Obtain RestTemplate	380
139.4. Invoke HTTP Call	380
139.5. Evaluate Response	381
140. WebClient Client	382
140.1. Obtain WebClient	382
140.2. Invoke HTTP Call	382
141. Implementing Parameters	384
141.1. Controller Parameter Handling	384
141.2. Client-side Parameter Handling	385
142. Accessing HTTP Responses	387
142.1. Obtaining ResponseEntity	387
142.2. ResponseEntity<T>	387
143. Client Error Handling	389
143.1. RestTemplate Response Exceptions	389
143.2. WebClient Response Exceptions	390

143.3. RestTemplate and WebClient Exceptions	390
144. Controller Responses	392
144.1. Controller Return ResponseEntity	392
144.2. Example ResponseEntity Responses	393
144.3. Controller Exception Handler	393
144.4. Simplified Controller Using ExceptionHandler	394
145. Summary	396
Controller/Service Interface	397
146. Introduction	398
146.1. Goals	398
146.2. Objectives	398
147. Roles	400
148. Error Reporting	401
148.1. Complex Object Result	401
148.2. Thrown Exception	401
148.3. Exceptions	402
148.4. Checked or Unchecked?	403
148.5. Candidate Client Exceptions	404
148.6. Service Errors	405
149. Controller Exception Advice	407
149.1. Service Method with Exception Logic	407
149.2. Controller Advice Class	408
149.3. Advice Exception Handlers	409
150. Summary	411
API Data Formats	412
151. Introduction	413
151.1. Goals	413
151.2. Objectives	413
152. Pattern Data Transfer Object	414
152.1. DTO Pattern Problem Space	414
152.2. DTO Pattern Solution Space	414
152.3. DTO Pattern Players	415
153. Sample DTO Class	417
154. Time/Date Detour	418
154.1. Pre Java 8 Time	418
154.2. java.time	418
154.3. Date/Time Formatting	419
154.4. Date/Time Exchange	420
155. Java Marshallers	422
156. JSON Content	423
156.1. Jackson JSON	423

156.2. JSON-B	426
157. XML Content	429
157.1. Jackson XML	430
157.2. JAXB	431
158. Configure Server-side Jackson	436
158.1. Dependencies	436
158.2. Configure ObjectMapper	436
158.3. Controller Properties	436
159. Client Marshall Request Content	439
160. Client Filters	441
160.1. RestTemplate	441
160.2. WebClient	442
161. Date/Time Lenient Parsing and Formatting	444
161.1. Out of the Box Time-related Formatting	444
161.2. Out of the Box Time-related Parsing	445
161.3. JSON-B DATE_FORMAT Option	446
161.4. JSON-B Custom Serializer Option	446
161.5. Jackson Lenient Parser	447
162. Summary	449
Swagger	450
163. Introduction	451
163.1. Goals	451
163.2. Objectives	451
164. Swagger Landscape	452
164.1. Open API Standard	452
164.2. Swagger-based Tools	452
164.3. Springfox	453
164.4. Springdoc	453
165. Minimal Configuration	455
165.1. Springfox Minimal Configuration	455
165.2. Springdoc Minimal Configuration	457
166. Example Use	459
166.1. Access Contest Controller POST Command	459
166.2. Invoke Contest Controller POST Command	460
166.3. View Contest Controller POST Command Results	460
167. Useful Configurations	462
167.1. Customizing Type Expressions	462
167.2. Duration Example Renderings	463
168. Springfox / Springdoc Analysis	467
169. Summary	468
Assignment 2 API	469

170. Overview	470
170.1. Grading Emphasis	470
170.2. Race Support	470
171. Assignment 2a: Modules	473
171.1. Purpose	473
171.2. Overview	473
171.3. Requirements	473
171.4. Grading	474
171.5. Additional Details	475
172. Assignment 2b: Content	476
172.1. Purpose	476
172.2. Overview	476
172.3. Requirements	477
172.4. Grading	478
172.5. Additional Details	478
173. Assignment 2c: Resources	480
173.1. Purpose	480
173.2. Overview	480
173.3. Requirements	481
173.4. Grading	482
173.5. Additional Details	482
174. Assignment 2d: Web Client/API Interactions	484
174.1. Purpose	484
174.2. Overview	484
174.3. Requirements	484
174.4. Grading	485
174.5. Additional Details	485
175. Assignment 2e: Service/Controller Interface	487
175.1. Purpose	487
175.2. Overview	487
175.3. Requirements	488
175.4. Grading	488
175.5. Additional Details	489
176. Assignment 2f: Required Test Scenarios	490
176.1. Scenario: Successful Registration	490
176.2. Requirements	491
176.3. Grading	492
Spring Security Introduction	493
177. Introduction	494
177.1. Goals	494
177.2. Objectives	494

178. Access Control	495
179. Privacy	496
179.1. Encoding	496
179.2. Encryption	496
179.3. Cryptographic Hash	497
180. Spring Web	499
181. No Security	500
181.1. Sample GET	500
181.2. Sample POST	500
181.3. Sample Static Content	501
182. Spring Security	503
182.1. Spring Core Authentication Framework	503
182.2. SecurityContext	505
183. Spring Boot Security AutoConfiguration	506
183.1. Maven Dependency	506
183.2. SecurityAutoConfiguration	506
183.3. UserDetailsServiceAutoConfiguration	507
183.4. SecurityFilterAutoConfiguration	507
184. Default FilterChain	508
185. Default Secured Application	509
185.1. Form Authentication Activated	509
185.2. Basic Authentication Activated	510
185.3. Authentication Required Activated	511
185.4. Username/Password Can be Supplied	511
185.5. CSRF Protection Activated	512
185.6. Other Headers	512
186. Default FilterChainProxy Bean	514
187. Summary	519
Spring Security Authentication	520
188. Introduction	521
188.1. Goals	521
188.2. Objectives	521
189. WebSecurityConfigurer	522
189.1. Core Application Security Configuration	522
189.2. Additional Swagger Security Configuration	523
189.3. Ignoring Static Resources	524
189.4. SecurityFilterChain Matcher	525
189.5. HttpSecurity Builder Methods	526
189.6. Authorize Requests	526
189.7. Authentication	527
189.8. Header Configuration	527

189.9. Stateless Session Configuration	528
190. Configuration Results	529
190.1. Successful Anonymous Call	529
190.2. Successful Authenticated Call	529
190.3. Rejected Unauthenticated Call Attempt	530
191. Authenticated User	531
191.1. Inject UserDetails into Call	531
191.2. Obtain SecurityContext from Holder	531
192. Swagger BASIC Auth Configuration	532
192.1. Swagger Authentication Configuration	532
192.2. Swagger Security Scheme	532
193. CORS	536
193.1. Origin Request Header	536
193.2. Access-Control-Allow-Origin Response Header	536
193.3. Browser Blocks Response	537
193.4. Browser Supplied Origin of Javascript	537
193.5. Spring MVC @CrossOrigin Annotation	538
193.6. Spring Security CORS Filter	538
193.7. Dynamically Permit All CORS Requests	539
193.8. Custom CORS Configuration	540
193.9. Successful Browser/Server CORS Exchange	540
194. RestTemplate Authentication	542
194.1. Authentication Integration Tests with RestTemplate	542
195. Mock MVC Authentication	544
195.1. MockMvc Anonymous Call	544
195.2. MockMvc Authenticated Call	545
196. Summary	546
User Details	547
197. Introduction	548
197.1. Goals	548
197.2. Objectives	548
198. AuthenticationManager	549
198.1. ProviderManager	549
198.2. AuthenticationManagerBuilder	550
198.3. AuthenticationProvider	551
198.4. AbstractUserDetailsAuthenticationProvider	551
198.5. DaoAuthenticationProvider	552
198.6. UserDetailsService	553
199. WebSecurityConfigurer	554
199.1. Directly Wire-up Parent AuthenticationManager	554
199.2. Directly Wire-up AuthenticationProvider	555

199.3. Use Local AuthenticationManagerBuilder	555
199.4. Define Service and Encoder @Bean	557
199.5. Combine Approaches	559
199.6. AuthenticationManager @Bean	561
200. UserDetails	562
201. PasswordEncoder	563
201.1. NoOpPasswordEncoder	563
201.2. BCryptPasswordEncoder	563
201.3. DelegatingPasswordEncoder	563
202. JDBC UserDetailsService	564
202.1. H2 Database	565
202.2. DataSource: Maven Dependencies	565
202.3. JDBC UserDetailsService	565
202.4. Inject/Add JDBC UserDetailsService	566
202.5. Autogenerated Database URL	566
202.6. Specified Database URL	566
202.7. Enable H2 Console Security Settings	567
202.8. Form Login	568
202.9. H2 Login	569
202.10. H2 Console	569
202.11. Create DB Schema Script	569
202.12. Schema Creation	570
202.13. Create User DB Populate Script	571
202.14. User DB Population	571
202.15. H2 User Access	572
202.16. Authenticate Access using JDBC UserDetailsService	572
202.17. Encrypting Passwords	573
203. Final Examples	575
203.1. Authenticate to All Three UserDetailsServicees	575
203.2. Authenticate to All Three Users	575
204. Summary	577
Authorization	578
205. Introduction	579
205.1. Goals	579
205.2. Objectives	579
206. Authorities, Roles, Permissions	580
207. Authorization Constraint Types	581
207.1. Path-based Constraints	581
207.2. Annotation-based Constraints	581
208. Setup	583
208.1. Who Am I Controller	583

208.2. Demonstration Users	584
208.3. Core FilterChainProxy Setup	584
208.4. Controller Operations	584
209. Path-based Authorizations	586
209.1. Path-based Role Authorization Constraints	586
209.2. Example Path-based Role Authorization (Sam)	586
209.3. Example Path-based Role Authorization (Woody)	587
210. Path-based Authority Permission Constraints	588
210.1. Path-based Authority Permission (Norm)	588
210.2. Path-based Authority Permission (Frasier)	588
210.3. Path-based Authority Permission (Sam and Woody)	589
210.4. Other Path Constraints	589
210.5. Other Path Constraints Usage	590
211. Authorization	591
211.1. Review: FilterSecurityInterceptor At End of Chain	591
211.2. FilterSecurityInterceptor Calls	591
211.3. AccessDecisionManager	592
211.4. Assigning Custom AccessDecisionManager	593
211.5. AccessDecisionVoter	594
212. Role Inheritance	595
212.1. Role Inheritance Definition	595
213. @Secure	596
213.1. Enabling @Secured Annotations	596
213.2. @Secured Annotation	596
213.3. @Secured Annotation Checks	597
213.4. @Secured Many Roles	597
213.5. @Secured Only Processing Roles	598
213.6. @Secured Does Not Support Role Inheritance	598
214. Controller Advice	600
214.1. AccessDeniedException Exception Handler	600
214.2. AccessDeniedException Exception Result	601
215. JSR-250	602
215.1. Enabling JSR-250	602
215.2. @RolesAllowed Annotation	602
215.3. @RolesAllowed Annotation Checks	602
215.4. Multiple Roles	603
215.5. Multiple Role Check	603
215.6. JSR-250 Does not Support Non-Role Authorities	603
216. Expressions	605
216.1. Expression Role Constraint	605
216.2. Expression Role Constraint Checks	605

216.3. Expressions Support Permissions and Role Inheritance	606
216.4. Supports Permissions and Boolean Logic	606
217. Summary	608
JWT/JWS Token Authn/Authz	609
218. Introduction	610
218.1. Goals	610
218.2. Objectives	610
219. Identity and Authorities	611
219.1. BASIC Authentication/Authorization	611
220. Tokens	613
220.1. Token Authentication/Login	613
220.2. Token Authorization/Operation	614
220.3. Authentication Separate from Authorization	615
220.4. JWT Terms	615
221. JWT Authentication	617
221.1. Example JWT Authentication/Login Flow	617
221.2. Example JWT Authorization/Operation Call Flow	617
222. Maven Dependencies	619
223. JwtConfig	620
223.1. JwtConfig application.properties	620
224. JwtUtil	622
224.1. Dependencies on JwtUtil	622
224.2. JwtUtil: generateToken()	623
224.3. JwtUtil: generateToken() Helper Methods	623
224.4. Example Encoded JWS	624
224.5. Example Decoded JWS Header and Body	624
224.6. JwtUtil: parseToken()	625
224.7. JwtUtil: parseToken() Helper Methods	626
225. JwtAuthenticationFilter	627
225.1. JwtAuthenticationFilter Relationships	627
225.2. JwtAuthenticationFilter: Constructor	628
225.3. JwtAuthenticationFilter: attemptAuthentication()	628
225.4. JwtAuthenticationFilter: attemptAuthentication() DTO	629
225.5. JwtAuthenticationFilter: attemptAuthentication() Helper Method	630
225.6. JwtAuthenticationFilter: successfulAuthentication()	630
226. JwtAuthorizationFilter	632
226.1. JwtAuthorizationFilter Relationships	632
226.2. JwtAuthorizationFilter: Constructor	633
226.3. JwtAuthorizationFilter: doFilterInternal()	633
226.4. JwtAuthenticationToken	634
226.5. JwtEntryPoint	636

227. API Security Configuration	637
227.1. API Authentication Manager Builder	637
227.2. API HttpSecurity Key JWS Parts	638
227.3. API HttpSecurity Full Details	639
228. Example JWT/JWS Application	640
228.1. Roles and Role Inheritance	640
228.2. CartsService	640
228.3. Login	641
228.4. createCart()	642
228.5. addItem()	643
228.6. getCart()	644
228.7. removeCart()	645
229. Summary	647
Enabling HTTPS	648
230. Introduction	649
230.1. Goals	649
230.2. Objectives	649
231. HTTP Access	650
232. HTTPS	651
232.1. HTTPS/TLS	651
232.2. Keystores	651
232.3. Tools	651
232.4. Self Signed Certificates	651
233. Enable HTTPS/TLS in Spring Boot	653
233.1. Generate Self-signed Certificate	653
233.2. Place Keystore in Reference-able Location	653
233.3. Add TLS properties	654
234. Untrusted Certificate Error	655
235. Accept Self-signed Certificates	656
235.1. Optional Redirect	656
235.2. HTTP:8080 ⇒ HTTPS:8443 Redirect Example	657
235.3. Follow Redirects	658
235.4. Caution About Redirects	658
236. Maven Integration Test	659
236.1. Maven Integration Test Phases	659
236.2. Spring Boot Maven Plugin	660
236.3. Build Helper Maven Plugin	662
236.4. Failsafe Plugin	662
236.5. JUnit @SpringBootTest	664
236.6. ClientTestConfiguration	664
236.7. application-its.properties	665

236.8. username/password Credentials	665
236.9. ServerConfig	666
236.10. authnUrl URI	666
236.11. authUser RestTemplate	666
236.12. HTTPS ClientHttpRequestFactory	667
236.13. SSL Context	667
236.14. JUnit @Test	668
236.15. Maven Verify	669
237. Summary	672
Assignment 3: Security	673
238. Assignment Starter	674
239. Assignment Support	675
240. Assignment 3a: Security Authentication	676
240.1. Anonymous Access	676
240.2. Authenticated Access	678
240.3. User Details	683
241. Assignment 3b: Security Authorization	686
241.1. Authorities	686
241.2. Authorization	687
241.3. HTTPS	690
242. Assignment 3c: AOP and Method Proxies	692
242.1. Reflection	692
242.2. Dynamic Proxies	694
242.3. Aspects	696
Spring AOP and Method Proxies	700
243. Introduction	701
243.1. Goals	701
243.2. Objectives	701
244. Rationale	702
244.1. Adding More Cross-Cutting Capabilities	702
244.2. Using Proxies	702
245. Reflection	704
245.1. Reflection Method	704
245.2. Calling Reflection Method	705
245.3. Reflection Method Result	705
246. JDK Dynamic Proxies	707
246.1. Creating Dynamic Proxy	707
246.2. Generated Dynamic Proxy Class Output	708
246.3. Alternative Proxy All Construction	708
246.4. InvocationHandler Class	708
246.5. InvocationHandler invoke() Method	709

246.6. Calling Proxied Object	710
247. CGLIB	711
247.1. Creating CGLIB Proxy	711
247.2. MethodInterceptor Class	712
247.3. MethodInterceptor intercept() Method	712
247.4. Calling CGLIB Proxied Object	713
248. Interpose	714
249. Spring AOP	715
249.1. AOP Definitions	715
249.2. Enabling Spring AOP	716
249.3. Aspect Class	717
249.4. Pointcut	717
249.5. Pointcut Expression	718
249.6. Example Pointcut Definition	719
249.7. Combining Pointcut Expressions	719
249.8. Advice	719
250. Pointcut Expression Examples	721
250.1. execution Pointcut Expression	721
250.2. within Pointcut Expression	723
250.3. target and this Pointcut Expressions	723
251. Advice Parameters	724
251.1. Typed Advice Parameters	724
251.2. Multiple,Typed Advice Parameters	725
251.3. Annotation Parameters	725
251.4. Target and Proxy Parameters	726
251.5. Dynamic Parameters	727
251.6. Dynamic Parameters Output	727
252. Advice Types	729
252.1. @Before	729
252.2. @AfterReturning	730
252.3. @AfterThrowing	730
252.4. @After	731
252.5. @Around	731
253. Other Features	733
254. Summary	734
Heroku Deployments	735
255. Introduction	736
255.1. Goals	736
255.2. Objectives	736
256. Heroku Background	737
257. Setup Heroku	738

258. Heroku Login	739
259. Create Heroku App	740
260. Create Spring Boot Application	741
260.1. Example Source Tree	741
260.2. Starting Example	742
260.3. Client Access	742
260.4. Local Unit Integration Test	742
261. Maven Heroku Deployment	744
261.1. Heroku Maven Plugin	744
261.2. Deployment appName	745
261.3. Example settings.xml Profile	746
261.4. Using Profiles	746
261.5. Maven Heroku Deploy Goal	747
261.6. Tail Logs	748
261.7. Access Site	748
261.8. Access Via Swagger	749
262. Remote IT Test	751
262.1. JUnit IT Test Case	751
262.2. IT Properties	752
262.3. Configuration	755
262.4. JUnit IT Test	759
262.5. Simple Communications Test	759
262.6. Authentication Test	759
262.7. Automation	760
263. Summary	763
Docker Images	764
264. Introduction	765
264.1. Goals	765
264.2. Objectives	765
265. Containers	766
265.1. Container Deployments	766
266. Docker Ecosystem	767
266.1. Container Builders	767
266.2. Container Runtimes	767
267. Docker Images	768
268. Basic Docker Image	769
268.1. Basic Dockerfile	769
268.2. Basic Docker Image Build Output	770
268.3. Local Docker Registry	770
268.4. Running Docker Image	771
268.5. Docker Run Command with Arguments	771

268.6. Running Docker Image	772
268.7. Using the Docker Image	772
268.8. Docker Image is Layered	772
268.9. Application Layer	773
268.10. Spring Boot Plugin	774
268.11. Building Docker Image using Buildpack	774
268.12. Buildpack Image in Local Docker Repository	775
268.13. Buildpack Image Execution	776
268.14. Inspecting Buildpack Image	776
269. Layers	778
269.1. Analyzing Basic Docker Image	778
269.2. Analyzing Basic Buildpack Image	779
270. Adding Fine-grain Layering	781
270.1. Configure Layer-ready Executable JAR	781
270.2. Building and Inspecting Layer-ready Executable JAR	781
270.3. Default Executable JAR Layers	782
271. Layered Buildpack Image	783
271.1. Dependency Layer	783
271.2. Snapshot Layer	783
271.3. Application Layer	784
271.4. Review: Single Layer Application	784
272. Layered Docker Image	786
272.1. Example Layered Dockerfile	786
272.2. Builder Phase	786
272.3. Construction Phase	787
272.4. Dependency Layer	788
272.5. Snapshot Layer	788
272.6. Application Layer	788
273. Summary	790
Heroku Docker Deployments	791
274. Introduction	792
274.1. Goals	792
274.2. Objectives	792
275. Heroku Docker Notes	793
276. Heroku Login	794
276.1. Heroku Container Login	794
276.2. Create Heroku App	794
277. Adjust Dockerfile	796
277.1. Test Dockerfile server.port	796
278. Deploy Docker Image	798
278.1. Deploying Tagged Image	798

278.2. Push using Heroku CLI	799
279. Complete Deployment	800
279.1. Release Pushed Image to Users	800
279.2. Tail Logs	800
279.3. Access Site	800
280. Summary	802
Docker Compose	803
281. Introduction	804
281.1. Goals	804
281.2. Objectives	804
282. Development and Integration Testing with Real Resources	805
282.1. Managing Images	805
283. Docker Compose	807
283.1. Docker Compose is Local to One Machine	807
284. Docker Compose Configuration File	808
284.1. mongo Service Definition	808
284.2. postgres Service Definition	809
284.3. api Service Definition	809
284.4. Build/Download Images	810
284.5. Default Port Assignments	811
284.6. Compose Override Files	811
284.7. Compose Override File Naming	812
284.8. Multiple Compose Files	813
284.9. Environment Files	813
285. Docker Compose Commands	815
285.1. Build Source Images	815
285.2. Start Services in Foreground	815
285.3. Project Name	815
285.4. Start Services in Background	816
285.5. Access Service Logs	816
285.6. Stop Running Services	817
286. Docker Cleanup	818
286.1. Docker Image Prune	819
286.2. Docker System Prune	819
286.3. Image Repository State After Pruning	820
287. Summary	821
Assignment 4: Deployments	822
288. Assignment 4a: Application Deployment Option	823
288.1. Purpose	823
288.2. Overview	823
288.3. Requirements	823

289. Assignment 4b: Docker Deployment Option.....	825
289.1. Docker Image	825
289.2. Heroku Docker Deploy.....	827
RDBMS.....	829
290. Introduction	830
290.1. Goals	830
290.2. Objectives	830
291. Schema Concepts	831
291.1. RDBMS Tables/Columns	831
291.2. Column Data	831
291.3. Column Types	831
291.4. Example Column Types	832
291.5. Constraints	832
291.6. Primary Key	833
291.7. UUID	834
291.8. Database Sequence	835
292. Example POJO	837
293. Schema	838
293.1. Schema Creation	838
293.2. Example Schema	838
294. Schema Command Line Population	840
294.1. Schema Result	840
294.2. List Tables	841
294.3. Describe Song Table	841
295. RDBMS Project	842
295.1. RDBMS Project Dependencies	842
295.2. RDBMS Access Objects	843
295.3. RDBMS Connection Properties	843
296. Schema Migration	845
296.1. Flyway Automated Schema Migration	845
296.2. Flyway Schema Source	845
296.3. Flyway Automatic Schema Population	845
296.4. Database Server Profiles	846
296.5. Dirty Database Detection	846
296.6. Flyway Migration	847
297. SQL CRUD Commands	848
297.1. H2 Console Access	848
297.2. Postgres CLI Access	848
297.3. Next Value for Sequence	849
297.4. SQL ROW INSERT	849
297.5. SQL SELECT	849

297.6. SQL ROW UPDATE	850
297.7. SQL ROW DELETE	851
297.8. RDBMS Transaction	851
298. JDBC	853
298.1. JDBC DataSource	853
298.2. Obtain Connection and Statement	853
298.3. JDBC Create Example	854
298.4. Set ID Example	855
298.5. JDBC Select Example	855
298.6. nextId	856
298.7. Dialect	857
299. Summary	858
Java Persistence API (JPA)	859
300. Introduction	860
300.1. Goals	860
300.2. Objectives	860
301. Java Persistence API	861
301.1. JPA Standard and Providers	861
301.2. JPA Dependencies	861
301.3. Enabling JPA AutoConfiguration	862
301.4. Configuring JPA DataSource	862
301.5. Automatic Schema Generation	863
301.6. Schema Generation to File	863
301.7. Other Useful Properties	864
301.8. Configuring JPA Entity Scan	865
301.9. JPA Persistence Unit	865
301.10. JPA Persistence Context	866
302. JPA Entity	867
302.1. JPA @Entity Defaults	867
302.2. JPA Overrides	868
303. Basic JPA CRUD Commands	869
303.1. EntityManager persist()	869
303.2. EntityManager find() By Identity	869
303.3. EntityManager query	870
303.4. EntityManager flush()	871
303.5. EntityManager remove()	871
303.6. EntityManager clear() and detach()	872
304. Transactions	873
304.1. Transactions Required for Explicit Changes/Actions	873
304.2. Activating Transactions	873
304.3. Conceptual Transaction Handling	874

304.4. Activating Transactions in @Components	875
304.5. Calling @Transactional @Component Methods	875
304.6. @Transactional @Component Methods SQL	876
304.7. Unmanaged @Entity	876
304.8. Shared Transaction	877
304.9. @Transactional Attributes	878
305. Summary	879
Spring Data JPA Repository	880
306. Introduction	881
306.1. Goals	881
306.2. Objectives	881
307. Spring Data JPA Repository	882
308. Spring Data Repository Interfaces	883
309. SongsRepository	884
309.1. Song @Entity	884
309.2. SongsRepository	884
310. Configuration	885
310.1. Injection	885
311. CrudRepository	886
311.1. CrudRepository save() New	886
311.2. CrudRepository save() Update Existing	887
311.3. CrudRepository save()/Update Resulting SQL	887
311.4. New Entity?	888
311.5. CrudRepository existsById()	888
311.6. CrudRepository findById()	889
311.7. CrudRepository delete()	890
311.8. CrudRepository deleteById()	891
311.9. Other CrudRepository Methods	891
312. PagingAndSortingRepository	893
312.1. Sorting	893
312.2. Paging	894
312.3. Page Result	895
312.4. Slice Properties	895
312.5. Page Properties	896
312.6. Stateful Pageable Creation	896
312.7. Page Iteration	897
313. Query By Example	898
313.1. Example Object	898
313.2. findAll By Example	899
313.3. Primitive Types are Non-Null	899
313.4. matchingAny ExampleMatcher	900

313.5. Ignoring Properties	901
313.6. Contains ExampleMatcher	901
314. Derived Queries	903
314.1. Single Field Exact Match Example	903
314.2. Query Keywords	904
314.3. Other Keywords	904
314.4. Multiple Fields	905
314.5. Collection Response Query Example	905
314.6. Slice Response Query Example	906
314.7. Page Response Query Example	907
315. JPA-QL Named Queries	909
315.1. Mapping @NamedQueries to Repository Methods	909
316. @Query Annotation Queries	911
316.1. @Query Annotation Native Queries	911
317. JpaRepository Methods	912
317.1. JpaRepository Type Extensions	912
317.2. JpaRepository flush()	912
317.3. JpaRepository deleteInBatch()	914
317.4. JPA References	914
318. Custom Queries	917
318.1. Custom Query Interface	917
318.2. Repository Extends Custom Query Interface	917
318.3. Custom Query Method Implementation	917
318.4. Repository Implementation Postfix	918
318.5. Helper Methods	918
318.6. Naive Injections	919
318.7. Required Injections	919
318.8. Calling Custom Query	920
319. Summary	921
319.1. Comparing Query Types	921
JPA Repository End-to-End Application	922
320. Introduction	923
320.1. Goals	923
320.2. Objectives	923
321. BO/DTO Component Architecture	924
321.1. Business Object(s)/@Entities	924
321.2. Data Transfer Object(s) (DTOs)	924
321.3. BO/DTO Mapping	925
322. Service Architecture	929
322.1. Injected Service Boundaries	929
322.2. Compound Services	931

323. BO/DTO Interface Options	933
323.1. API Maps DTO/BO	933
323.2. @Service Maps DTO/BO	933
323.3. Layered Service Mapping Approach	934
324. Implementation Details	935
324.1. Song BO	935
324.2. SongDTO	935
324.3. Song JSON Rendering	937
324.4. Song XML Rendering	937
324.5. Pageable/PageableDTO	938
324.6. Page/PageDTO	940
325. SongMapper	943
325.1. Example Map: SongDTO to Song BO	943
325.2. Example Map: Song BO to SongDTO	943
326. Service Tier	944
326.1. SongsService Interface	944
326.2. SongsServiceImpl Class	944
326.3. createSong()	945
326.4. findSongsMatchingAll()	945
327. RestController API	947
327.1. createSong()	947
327.2. findSongsByExample()	948
327.3. WebClient Example	948
328. Summary	950
MongoDB with Mongo Shell	951
329. Introduction	952
329.1. Goals	952
329.2. Objectives	952
330. Mongo Concepts	953
330.1. Mongo Terms	953
330.2. Mongo Documents	953
331. MongoDB Server	955
331.1. Starting Docker-Compose MongoDB	955
331.2. Connecting using Host's Mongo Shell	955
331.3. Connecting using Guest's Mongo Shell	955
331.4. Switch to test Database	956
331.5. Database Command Help	956
332. Basic CRUD Commands	958
332.1. Insert Document	958
332.2. Primary Keys	958
332.3. Document Index	958

332.4. Create Index	959
332.5. Find All Documents	959
332.6. Return Only Specific Fields	960
332.7. Get Document by Id	960
332.8. Replace Document	960
332.9. Save/Upsert a Document	961
332.10. Update Field	961
332.11. Delete a Document	962
333. Paging Commands	963
333.1. Sample Documents	963
333.2. limit()	963
333.3. sort()/skip()/limit()	964
334. Aggregation Pipelines	966
334.1. Common Commands	966
334.2. Unique Commands	966
334.3. Simple Match Example	966
334.4. Count Matches	967
335. Helpful Commands	969
335.1. Default Database	969
335.2. Command-Line Script	969
336. Summary	971
MongoTemplate	972
337. Introduction	973
337.1. Goals	973
337.2. Objectives	973
338. Mongo Project	975
338.1. Mongo Project Dependencies	975
338.2. Mongo Project Integration Testing Options	975
338.3. Flapdoodle Test Dependencies	976
338.4. MongoDB Access Objects	976
338.5. MongoDB Connection Properties	977
338.6. Injecting MongoTemplate	978
338.7. Disabling Embedded MongoDB	978
338.8. @ActiveProfiles	979
338.9. TestProfileResolver	979
338.10. Using TestProfileResolver	980
338.11. Inject MongoTemplate	980
339. Example POJO	982
339.1. Property Mapping	982
339.2. Field Mapping	983
339.3. Instantiation	984

339.4. Property Population	984
340. Command Types	985
341. Whole Document Operations	986
341.1. insert()	986
341.2. save()/Upsert	987
341.3. remove()	988
342. Operations By ID	989
342.1. findById()	989
343. Operations By Query Filter	990
343.1. exists() By Criteria	990
343.2. delete()	991
344. Field Modification Operations	992
344.1. update() Field(s)	992
344.2. upsert() Fields	992
345. Paging	994
345.1. skip() / limit()	994
345.2. Sort	994
345.3. Pageable	995
346. Aggregation	996
347. ACID Transactions	997
347.1. Atomicity	997
347.2. Consistency	997
347.3. Isolation	997
347.4. Durability	998
348. Summary	999
Spring Data MongoDB Repository	1000
349. Introduction	1001
349.1. Goals	1001
349.2. Objectives	1001
350. Spring Data MongoDB Repository	1002
351. Spring Data MongoDB Repository Interfaces	1003
352. BooksRepository	1004
352.1. Book @Document	1004
352.2. BooksRepository	1004
353. Configuration	1005
353.1. Injection	1005
354. CrudRepository	1006
354.1. CrudRepository save() New	1006
354.2. CrudRepository save() Update Existing	1006
354.3. CrudRepository save() / Update Resulting Mongo Command	1007
354.4. CrudRepository existsById()	1007

354.5. CrudRepository findById()	1008
354.6. CrudRepository delete()	1009
354.7. CrudRepository deleteById()	1010
354.8. Other CrudRepository Methods	1010
355. PagingAndSortingRepository	1011
355.1. Sorting	1011
355.2. Paging	1012
355.3. Page Result	1013
355.4. Slice Properties	1013
355.5. Page Properties	1014
355.6. Stateful Pageable Creation	1014
355.7. Page Iteration	1014
356. Query By Example	1015
356.1. Example Object	1015
356.2. findAll By Example	1016
356.3. Ignoring Properties	1016
356.4. Contains ExampleMatcher	1017
357. Derived Queries	1018
357.1. Single Field Exact Match Example	1018
357.2. Query Keywords	1019
357.3. Other Keywords	1019
357.4. Multiple Fields	1020
357.5. Collection Response Query Example	1020
357.6. Slice Response Query Example	1021
357.7. Page Response Query Example	1021
358. @Query Annotation Queries	1023
358.1. @Query Annotation Attributes	1023
359. MongoRepository Methods	1024
360. Custom Queries	1025
360.1. Custom Query Interface	1025
360.2. Repository Extends Custom Query Interface	1025
360.3. Custom Query Method Implementation	1025
360.4. Repository Implementation Postfix	1026
360.5. Helper Methods	1026
360.6. Naive Injections	1027
360.7. Required Injections	1027
360.8. Calling Custom Query	1028
360.9. Implementing Aggregation	1028
361. Summary	1029
Mongo Repository End-to-End Application	1030
362. Introduction	1031

362.1. Goals	1031
362.2. Objectives	1031
363. BO/DTO Component Architecture	1032
363.1. Business Object(s)/@Documents	1032
363.2. Data Transfer Object(s) (DTOs)	1033
363.3. BookDTO Class	1033
363.4. BO/DTO Mapping	1034
364. Service Architecture	1037
364.1. Injected Service Boundaries	1037
364.2. Compound Services	1039
365. BO/DTO Interface Options	1041
365.1. API Maps DTO/BO	1041
365.2. @Service Maps DTO/BO	1041
365.3. Layered Service Mapping Approach	1042
366. Implementation Details	1043
366.1. Book BO	1043
366.2. BookDTO	1043
366.3. Book JSON Rendering	1045
366.4. Book XML Rendering	1045
366.5. Pageable/PageableDTO	1046
366.6. Page/PageDTO	1048
367. BookMapper	1051
367.1. Example Map: BookDTO to Book BO	1051
367.2. Example Map: Book BO to BookDTO	1051
368. Service Tier	1052
368.1. BooksService Interface	1052
368.2. BooksServiceImpl Class	1052
368.3. createBook()	1053
368.4. findBooksMatchingAll()	1053
369. RestController API	1055
369.1. createBook()	1055
369.2. findBooksByExample()	1056
369.3. WebClient Example	1056
370. Summary	1058
Heroku Database Deployments	1059
371. Introduction	1060
371.1. Goals	1060
371.2. Objectives	1060
372. Production Properties	1061
372.1. Postgres Production Properties	1061
372.2. Mongo Production Properties	1061

373. Parsing Runtime Properties	1062
373.1. Environment Variable Script	1062
373.2. Script Output	1063
373.3. Heroku DataSource Property	1063
373.4. Testing DATABASE_URL	1064
373.5. MongoDB Properties	1065
373.6. PORT Property	1065
374. Docker Image	1066
374.1. Dockerfile	1066
374.2. Spotify Docker Build Maven Plugin	1066
375. Heroku Deployment	1068
375.1. Provision MongoDB	1068
375.2. Provision Application	1068
375.3. Provision Postgres	1068
375.4. Deploy Application	1069
375.5. Release the Application	1069
376. Summary	1071
Assignment 5: DB	1072
377. Assignment 5a: Spring Data JPA	1073
377.1. Database Schema	1073
377.2. Entity/BO Class	1077
377.3. JPA Repository	1081
378. Assignment 5b: Spring Data Mongo	1084
378.1. Mongo Client Connection	1084
378.2. Mongo Document	1086
378.3. Mongo Repository	1089
379. Assignment 5c: Spring Data Application	1092
379.1. API/Service/DB End-to-End	1092
380. Assignment 5d: Bonus	1095
380.1. Races/Racers Alternate Repository	1095
Bean Validation	1097
381. Introduction	1098
381.1. Goals	1098
381.2. Objectives	1098
382. Background	1100
383. Dependencies	1101
384. Declarative Constraints	1102
384.1. Data Constraints	1102
384.2. Common Built-in Constraints	1102
384.3. Method Constraints	1103
385. Programmatic Validation	1104

385.1. Manual Validator Instantiation	1104
385.2. Inject Validator Instance	1104
385.3. Customizing Injected Instance	1105
385.4. Review: Class with Constraint	1105
385.5. Validate Object	1105
385.6. Validate Method Calls	1106
385.7. Identify Method Using Java Reflection	1107
385.8. Programmatically Check for Parameter Violations	1108
385.9. Validate Method Results	1108
386. Method Parameter Naming	1110
386.1. Add -parameters to Java Compiler Command	1110
386.2. Add Custom ParameterNameProvider	1111
386.3. ParameterNameProvider	1111
386.4. Named Parameters	1112
386.5. Determining Parameter Name	1113
387. Graphs	1114
387.1. Graph Non-Traversal	1114
387.2. Graph Traversal	1115
388. Groups	1116
388.1. Custom Validation Groups	1116
388.2. Applying Groups	1116
388.3. Skipping Groups	1117
388.4. Applying Groups	1117
389. Multiple Groups	1119
389.1. Example Class with Different Groups	1119
389.2. Validate All Supplied Groups	1119
389.3. Short-Circuit Validation	1120
389.4. Override Default Group	1121
390. Spring Integration	1122
390.1. Validated Component	1122
390.2. ConstraintViolationException	1123
390.3. Successful Validation	1123
390.4. Liskov Substitution Principle	1124
390.5. Disabling Parameter Constraint Override	1124
390.6. Spring Validated Group(s)	1125
390.7. Spring Validated Group(s) Example	1125
391. Custom Validation	1127
391.1. Constraint Interface Definition	1127
391.2. @Documented Annotation	1127
391.3. @Target Annotation	1128
391.4. @Retention	1129

391.5. @Repeatable	1130
391.6. @Constraint	1131
391.7. @MinAge-specific Properties	1132
391.8. Constraint Implementation Class	1133
391.9. Constraint Implementation Type Examples	1133
391.10. Constraint Initialization	1134
391.11. Constraint Validation	1134
391.12. Custom Violation Messages	1135
392. Cross-Parameter Validation	1137
392.1. Cross-Parameter Annotation	1137
392.2. @SupportedValidationTarget	1137
392.3. Method Call Correctness Validation	1138
392.4. Constraint Validation	1138
393. Web API Integration	1140
393.1. Vanilla Spring/AOP Validation	1140
393.2. ConstraintViolationException Not Handled	1140
393.3. ConstraintViolationException Exception Advice	1141
393.4. ConstraintViolationException Mapping Result	1142
393.5. Controller Constraint Validation	1142
393.6. MethodArgumentNotValidException	1143
393.7. MethodArgumentNotValidException Custom Mapping	1143
393.8. @PathVariable Validation	1145
393.9. @PathVariable Validation Result	1145
393.10. @RequestParam Validation	1146
393.11. @RequestParam Validation Violation Response	1146
393.12. Non-Client Errors	1147
393.13. Service Method Error	1147
393.14. Violation Incorrectly Reported as Client Error	1148
393.15. Checking Violation Source	1148
393.16. Internal Server Error Correctly Reported	1149
393.17. Service-detected Client Errors	1149
393.18. Payload	1150
393.19. Exception Handler Checking Payloads	1151
393.20. Internal Violation Exception Handler Results	1151
394. JPA Integration	1153
395. Mongo Integration	1154
395.1. Validating Saves	1154
395.2. ValidatingMongoEventListener	1155
395.3. Other AbstractMongoEventListener Events	1156
395.4. MongoMappingEvent	1156
396. Patterns / Anti-Patterns	1157

396.1. Data Tier Validation	1157
396.2. Use case-specific Validation	1157
396.3. Anti: Validation Everywhere	1158
397. Summary	1160
Integration Unit Testing	1161
398. Introduction	1162
398.1. Goals	1162
398.2. Objectives	1163
399. Votes and Elections Service	1164
399.1. Main Application Flows	1164
399.2. Service Event Integration	1164
400. Physical Architecture	1166
400.1. Integration Unit Test Physical Architecture	1166
401. Mongo Integration	1167
401.1. MongoDB Maven Dependencies	1167
401.2. Test MongoDB Maven Dependency	1167
401.3. MongoDB Properties	1168
401.4. MongoDB Repository	1168
401.5. VoteDTO MongoDB Document Class	1168
401.6. Sample MongoDB/VoterRepository Calls	1169
402. ActiveMQ Integration	1171
402.1. ActiveMQ Maven Dependencies	1171
402.2. ActiveMQ Integration Unit Test Properties	1171
402.3. Service Joinpoint Advice	1172
402.4. JMS Publish	1172
402.5. ObjectMapper	1173
402.6. JMS Receive	1174
402.7. EventListener	1174
403. JPA Integration	1176
403.1. JPA Core Maven Dependencies	1176
403.2. JPA Test Dependencies	1176
403.3. JPA Properties	1176
403.4. Database Schema Migration	1177
403.5. Flyway RDBMS Schema Migration	1177
403.6. Flyway RDBMS Schema Migration Files	1178
403.7. Flyway RDBMS Schema Migration Output	1179
403.8. JPA Repository	1179
403.9. Example VoteBO Entity Class	1179
403.10. Sample JPA/ElectionRepository Calls	1180
404. Unit Integration Test	1182
404.1. ClientTestConfiguration	1182

404.2. Example Test	1183
405. Summary.....	1184
Docker Compose Integration Testing	1185
406. Introduction	1186
406.1. Goals.....	1186
406.2. Objectives	1187
407. Integration Testing with Real Resources.....	1188
407.1. Managing Images	1188
408. Docker Compose Configuration File.....	1190
408.1. mongo Service Definition	1190
408.2. postgres Service Definition	1190
408.3. activemq Service Definition	1191
408.4. api Service Definition.....	1191
408.5. Compose Override Files.....	1192
409. Test Drive	1193
409.1. Clean Starting State.....	1193
409.2. Cast Two Votes	1193
409.3. Observe Updated State.....	1194
410. Inspect Images	1195
410.1. Exec Mongo CLI	1195
410.2. Exec Postgres CLI	1195
410.3. Exec Impact.....	1196
411. Integration Test Setup	1197
411.1. Integration Properties	1197
411.2. Maven Build Helper Plugin.....	1198
411.3. Maven Docker Compose Plugin	1198
411.4. Maven Docker Compose Plugin Output.....	1199
411.5. Maven Failsafe Plugin	1200
411.6. IT Test Client Configuration	1201
411.7. Example Failsafe Output	1201
411.8. IT Test Setup	1202
411.9. Wait For Services Startup	1203
412. Summary.....	1204
Testcontainers.....	1205
413. Introduction	1206
413.1. Goals.....	1206
413.2. Objectives	1206
414. Testcontainers Overview	1207
415. Example.....	1208
415.1. Maven Dependencies	1208
415.2. Main Tree	1208

415.3. Test Tree	1209
416. Example: Main Tree Artifacts	1210
416.1. Docker Compose File	1210
416.2. Docker Compose File Reference.....	1210
416.3. DockerComposeContainer.....	1211
416.4. Obtaining Runtime Port Numbers	1212
417. Example: Test Tree Artifacts	1213
417.1. Primary NTest Setup.....	1213
417.2. Injecting Dynamically Assigned Port#s.....	1214
417.3. DynamicPropertySource	1215
417.4. Injections Complete prior to Tests.....	1215
418. Exec Commands	1217
418.1. Exec MongoDB Command Output.....	1217
418.2. Exec Postgres Command Output	1218
419. Connect to Resources	1219
419.1. Maven Dependencies	1219
419.2. Injected Clients	1219
419.3. URL Templates	1220
419.4. Providing Dynamic Resource URL Declarations	1220
419.5. Application Properties	1221
419.6. JMS Listener	1221
419.7. Obtain Client Status	1222
419.8. Client Status Output	1223
420. Summary	1224
Testcontainers with Spock	1225
421. Introduction	1226
421.1. Goals	1226
421.2. Objectives	1227
422. Background	1228
422.1. Application Background	1228
422.2. Integration Testing Approach.....	1228
422.3. Docker Compose	1229
422.4. Testcontainers	1229
423. Docker Compose	1231
423.1. Docker Compose File	1231
423.2. Start Network	1232
423.3. Access Logs	1232
423.4. Execute Commands.....	1233
423.5. Shutdown Network	1233
423.6. Override/Extend Docker Compose File	1233
423.7. Using Mapped Host Ports	1234

423.8. Supplying Properties	1235
423.9. Specifying an Override File	1235
423.10. Override File Result	1236
424. Testcontainers and Spock	1237
424.1. Source Tree	1237
424.2. @SpringBootConfiguration	1238
424.3. Traditional @Bean Factories	1239
424.4. DockerComposeContainer	1239
424.5. @SpringBootTest	1240
424.6. Spock Network Management	1241
424.7. Set System Property	1242
424.8. ApplicationContextInitializer	1242
424.9. DynamicPropertySource	1243
424.10. Resulting Test Initialization Output	1244
425. Additional Waiting	1246
426. Executing Commands	1247
426.1. Example Command Output	1247
427. Client Connections	1249
427.1. Maven Dependencies	1249
427.2. Hard Coded Application Properties	1249
427.3. Dynamic URL Helper Methods	1250
427.4. Adding Dynamic Properties	1250
427.5. Adding JMS Listener	1251
427.6. Injecting Resource Clients	1252
427.7. Resource Client Calls	1252
428. Test Hierarchy	1253
428.1. Network Helper Class	1253
428.2. Integration Spec Base Class	1253
428.3. Specialized Integration Test Classes	1254
428.4. Test Execution Results	1254
429. Summary	1256

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 © 2021 jim stafford (jim.stafford@jhu.edu)

Chapter 2. Course Description

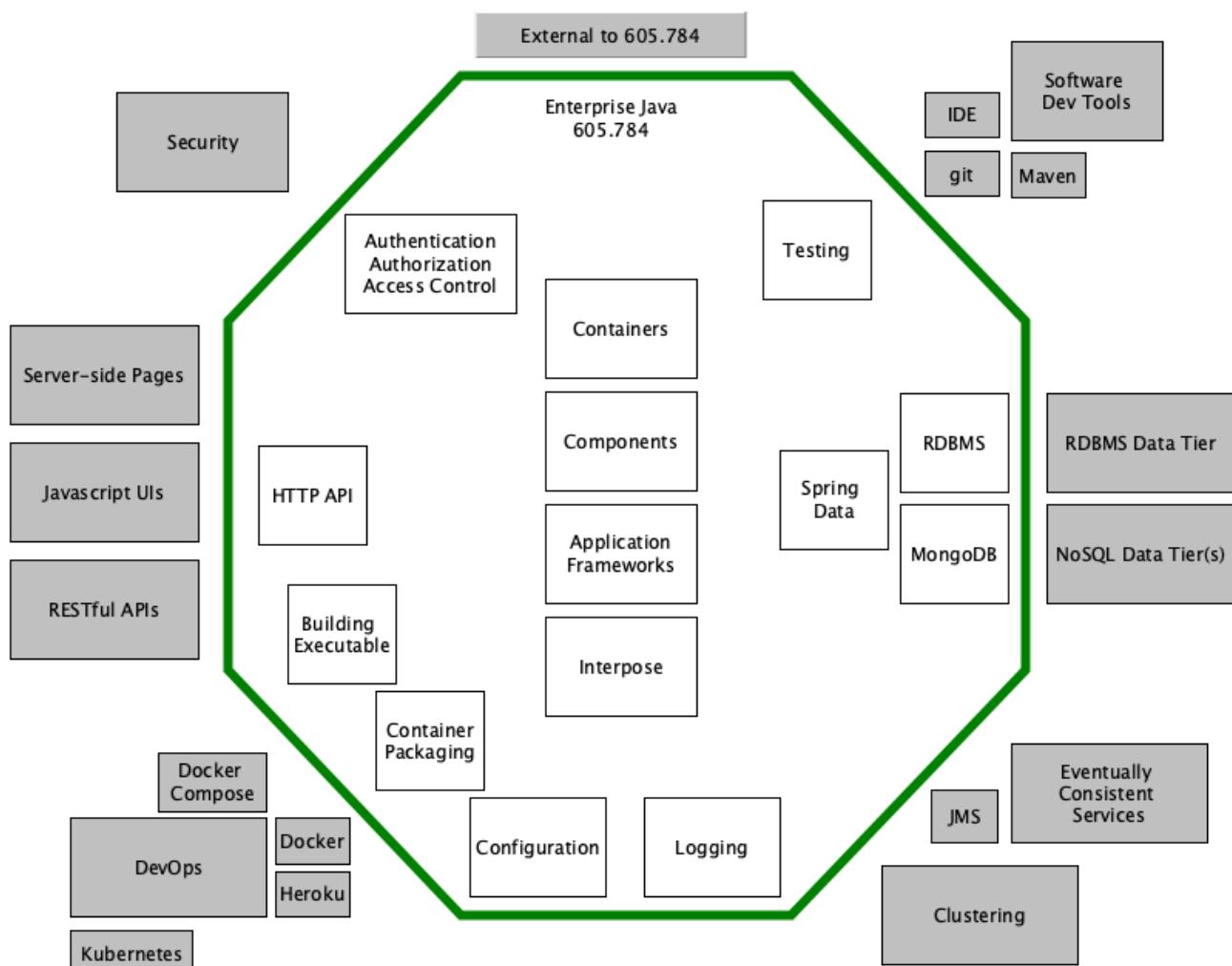
2.1. Meeting Times/Location

- Wednesdays, 4:30-7:10pm EST
 - via Zoom [Meeting ID: 961 7766 1958](#)
 - [Blackboard](#)

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 basic API access to service logic using modern RESTful approaches that include JSON and XML
- Implement basic data access tiers to relational and NoSQL (Mongo) 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.

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 11
 - c. Maven 3 (>= 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
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. JUnit
 - b. SLF/Logback
 - c. a relational database (H2 Database Engine) and JPA persistence provider (Hibernate)
 - d. application framework (Spring Boot 2.x, Spring 5.x).

2.8. Course Structure

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, data tier, and async processing. Each group consists of multiple topics that span multiple weeks.

The examples are available in a Github 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. 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.

2.9. Grading

- $100 \geq A \geq 90 > B \geq 80 > C \geq 70 > F$

Assessment	% of Semester Grade
Class/Newsgroup Participation	10% (9pm EST, Wed weekly cut-off)
Assignment 0: Application Build	5% (##)
Assignment 1: Application Config	15%
Assignment 2: Web API	15%
Assignment 3: Security	15%
Assignment 4: Deployment	10%
Assignment 5: Database	20%
Assignment 6: Async Methods	10% (###)



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 artifacts between the student and instructor.
 - ### Assignment 6 points will be merged with Assignment 5 if time constraints do not allow for a separate assignment
- 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 public 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 one 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—minus error. A link to the recording will be posted on blackboard.

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.

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: [Blackboard 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 an archive in GoogleDocs, or a link to a branch in a **private** repository.

2.14. Office Hours

Students needing further assistance are also welcome make other arrangements during the week or 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 the instructor and grader(s) with source code in a standard archive file. Links to archives on cloud storage acceptable.
- 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 or other Java-based unit test (e.g., Spock) frameworks that will run within the Maven surefire and failsafe environments.

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, 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...")

You will e-mail the projects to the grader and instructor with the following subject line

- (your name) project #; revision 0; part # of #

Your submission will include source archive (link to cloud storage is acceptable) and README (could be in source archive). The easiest way to do this is to archive the assignment from the root after completing a `mvn clean` build from the root.

If you need to make a correction, the correction should have the following e-mail subject. The body should describe what you wish to revise.

- (your name) project #; revision N; part # of #

Submission e-mails ([mail to all](#)):

- Jim - jim.stafford@jhu.edu

Chapter 4. Syllabus

Table 1. Core Development

#	Date	Lectures	Assignments/Notes
1	Sep01	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 Sep07, 6am
2	Sep08	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 Sep29, 6am
3	Sep15	<ul style="list-style-type: none"> Auto-Configuration notes 	
		Logging notes	<ul style="list-style-type: none"> Assignment 1b Logging spec Due: Wed Sep29, 6am
4	Sep22	Testing notes	<ul style="list-style-type: none"> Assignment 1c Testing spec Due: Wed Sep29, 6am

Table 2. Service and API Tiers

#	Date	Lectures	Assignments/Notes
4	Sep22 (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 Oct17, 8am
5	Sep29	API (Cont) <ul style="list-style-type: none"> Spring MVC notes Controller/Service Interface notes 	

#	Date	Lectures	Assignments/Notes
6	Oct06	<ul style="list-style-type: none"> • Data/Content Marshalling notes • API Documentation notes 	
7	Oct13	<p>Spring Security</p> <ul style="list-style-type: none"> • Spring Security Introduction notes • Authentication notes 	<ul style="list-style-type: none"> • no live class - recorded lectures • Assignment 3 Security and AOP spec Due: Sun Nov07, 6am
8	Oct20	<ul style="list-style-type: none"> • User Details notes • Access Control notes • [JSON Web Tokens notes] * optional topic • Enabling HTTPS/TLS notes 	
9	Oct27	AOP and Method Proxies notes	

Table 3. Data Tier

#	Date	Lectures	Assignments/Notes
10	Nov03	<p>Containers and Deployments</p> <ul style="list-style-type: none"> • Heroku Spring Boot Deployment notes • Docker Images notes • Heroku Docker Deployment notes • Docker Compose notes 	<ul style="list-style-type: none"> • Assignment 4 Deployment spec Due: Sun Nov21, 6am
11	Nov10	<p>JPA Mapping</p> <ul style="list-style-type: none"> • RDBMS notes • Java Persistence API (JPA) notes • Spring Data JPA Repository notes • Spring Data JPA End-to-End notes 	<ul style="list-style-type: none"> • Assignment 5a Spring Data JPA spec Due: Sun, Dec12, 6am

#	Date	Lectures	Assignments/Notes
12	Nov17	MongoDB NoSQL Mapping <ul style="list-style-type: none"> • MongoDB and Shell notes • MongoTemplate notes • Spring Data MongoDB Repository notes • Spring Data MongoDB End-to-End notes 	<ul style="list-style-type: none"> • Assignment 5b <p>Spring Data Mongo spec Due: Sun Dec12, 6am</p>
	Nov24	Thanksgiving	no class
13	Dec01		<ul style="list-style-type: none"> • Assignment 5c <p>Spring Data End-to-End spec Due: Sun Dec12, 6am</p>
14	Dec08	Heroku Database Deployments notes	
		Validation notes	

Table 4. Other Topics

		Async Processing	
		Integration Test Topics <ul style="list-style-type: none"> • Integration Unit Tests notes • Docker Compose IT notes • Testcontainers with JUnit notes • [Testcontainers with Spock] * optional topic notes 	

** points for Assignment 6 will be rolled into Assignment 5 if eliminated due to time constraints

Development Environment Setup

- Changes
 - 2021-08-26 - added docker-compose test drive of DB services
 - 2021-08-26 - added weekly course example git update commands

Chapter 5. Introduction

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

Software versions do not have to be latest-and-greatest. My JDK 11/Maven environment looks to be close to 2 years old when authoring this guide. For the most part, the firmest requirement is that the JDK must be 11 or at least your source code needs to stick to Java 11 features to be portable to grading environments.

You must manually download and install some of the software locally (e.g., IDE). Some have options (e.g., Docker/Mongo, Mongo). 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 11
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 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 11 compiler and its accompanying JRE environment immediately in class. Everything we do will revolve around a JVM.

Download and install a version of JDK 11. Candidate sources provided.

- Mac Users - [AdoptJDK](#) or thru brew
- Linux Users - package manager (e.g., yum, apt)

```
$ sudo apt install openjdk-11-jdk
```

- Windows Users - [AdoptJDK](#)



I have JDK installed thru brew on MacOS. Looking at the output, mine is a bit dated.

Example Java Version Check

```
$ java -version
openjdk version "11.0.4" 2019-07-16
OpenJDK Runtime Environment AdoptOpenJDK (build 11.0.4+11)
OpenJDK 64-Bit Server VM AdoptOpenJDK (build 11.0.4+11, mixed mode)

$ javac -version
javac 11.0.4
```

6.2. 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 through brew on MacOS. My version was released in 2019—so anything fairly recent should be good.

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 3.6.3 (cecedd343002696d0abb50b32b541b8a6ba2883f)
Maven home: /usr/local/Cellar/maven/3.6.3/libexec
Java version: 11.0.4, vendor: AdoptOpenJDK, runtime:
/Library/Java/JavaVirtualMachines/adoptopenjdk-11.jdk/Contents/Home
Default locale: en_US, platform encoding: UTF-8
OS name: "mac os x", version: "10.16", 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">

<!--
  <localRepository>somewhere_else</localRepository>
-->
<offline>false</offline>

<profiles>
</profiles>

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

</settings>
```

6.3. 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.

Download and install a Git Client.

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



I have git installed thru brew on MacOS

Example git Version Check

```
$ git --version  
git version 2.26.2
```

Checkout the course baseline.

```
$ git clone https://github.com/ejavaguy/ejava-springboot.git  
...  
$ ls | sort  
app  
assignment-starters  
build  
common  
coursedocs  
env  
intro  
pom.xml  
...
```

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

```
$ pwd  
.../ejava-springboot  
$ mvn clean install  
...
```

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

```
$ git checkout master # switches to master 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 in a 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 master  #switch back to course baseline  
$ git pull
```

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. It is free and works well. However, for this course I will actively be using IntelliJ IDEA. The community edition is free and contains most of the needed support.

Download and install an IDE for Java development.

- IntelliJ IDEA Community Edition
 - All platforms - [Jetbrains IntelliJ](#)
- Eclipse/STS
 - All platforms - [Spring.io](#)

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 web API client that can issue POST, PUT, and DELETE commands in addition to GET commands over HTTP and (one-way TLS) HTTPS. 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 simply be a browser plugin tool. However, the free version works and seems to only require a sign-up login.

```
$ 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)

It seems everything in this world has become containerized—and for a good reason. 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 tentative in requiring Docker for the class. I will make it optional for the students who cannot install. I will leverage Docker more heavily if I get a sense that all students have access. Let me know where you stand on this optional install.

Optionally download and install Docker. 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

Without Docker installation, you will

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

All platforms - [Docker.com](https://www.docker.com)

- Also install - [docker-compose](#)

 I was reading the docker-compose web page and noticed they are going through a breaking release. It should not break anything we have in source form. It should only impact pre-built images that have been downloaded. I will be investigating further and will update these notes.

Example Docker Version Check

```
$ docker -v  
Docker version 20.10.7, build f0df350  
$ docker-compose -v  
docker-compose version 1.29.2, build 5becea4c
```

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 -f env/docker-compose.yml -p ejava up -d ①②③  
Creating network "ejava_default" with the default driver  
Creating ejava_mongodb_1 ... done  
Creating ejava_postgres_1 ... done
```

① -f option references a configuration file to use

② -p option sets the project name to a well-known value (directory name is default)

③ up starts services and -d runs them all in the background

Shutdown database resources

```
ejava-springboot$ docker-compose -f env/docker-compose.yml -p ejava down
Stopping ejava_postgres_1 ... done
Stopping ejava_mongodb_1 ... done
Removing ejava_postgres_1 ... done
Removing ejava_mongodb_1 ... done
Removing network ejava_default
```

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. Note that you will eventually need an Internet accessible MongoDB instance later in the course, so feel free to activate your free Atlas account at any time.

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

- All platforms - [MongoDB](#)

6.8. Heroku Account (later)

Mid-way through the course you will hit a very exciting point in the course, where you will begin deploying your assignments to the Internet for all to see.

We will be leveraging the Heroku Internet hosting platform. Heroku supports deploying Spring Boot executable JARs as well as Docker images. You will need an account and download their "toolbelt" set of commands for uploading, configuring, and managing deployments.

Create an account and download the Heroku toolbelt.

- All platforms - [Heroku](#)

6.9. Mongo Atlas Account (later)

In the last 1/3 of the course, when deploying an application based on RDBMS and MongoDB, you will need access to an Internet accessible RDBMS and MongoDB instance. You will be able to provision a free RDBMS database directly from Heroku. You will be able to provision a free, Internet accessible MongoDB instance via Mongo Atlas.

Create an account and provision an Internet accessible MongoDB.

All platforms - [Mongo Atlas](#)

Introduction to Enterprise Java Frameworks

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 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 do with **control**.

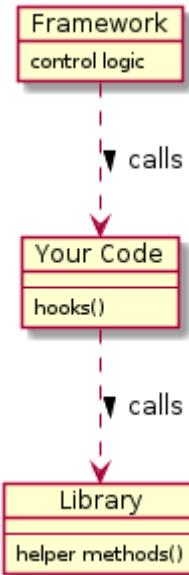


Figure 1. Library/ Framework/Code Relationship ^[2]

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)

It's 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.

[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.^[3] 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.*"

[3] "[Software framework](#)", Wikipedia

Chapter 10. Framework Enablers

10.1. Dependency Injection

A process to enable Inversion of Control (IoC), whereby objects define their dependencies ^[4] 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 of 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. ^[4]

— 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. ^[5]

— 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. ^[6]

— Spring.io, Container Overview

10.6. Interpose

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.

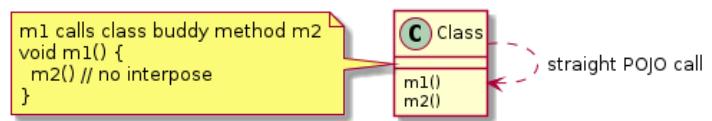


Figure 2. POJO Buddy Call

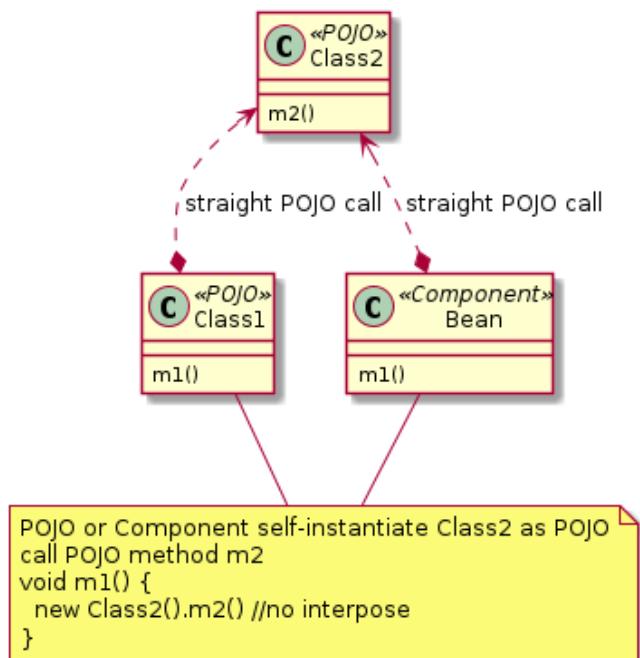


Figure 3. Self-Instantiated POJO Call

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.

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.

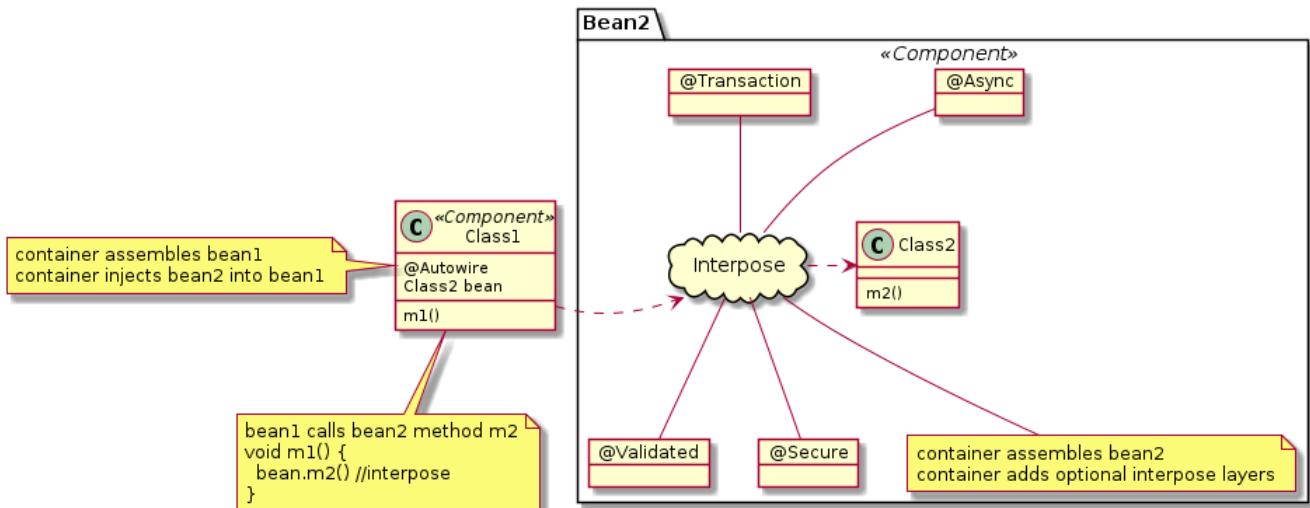


Figure 4. Container Interpose

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.

- [4] ["Spring Framework Documentation, The IoC Container"](#), Spring.io
- [5] ["Use containers to Build, Share and Run your applications"](#), Docker.com
- [6] ["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.

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, the 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.^[7] 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.^[8]

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.^[9] 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 framework seems to have paused for a while for naming changes and compatibility releases.^[10]

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.^[11]

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. ^[12]

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. ^[13] 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. ^[14]

- 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.^[15]

- 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.

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

[8] "Common Gateway Interface", Wikipedia

[9] "Jakarta EE", Wikipedia

[10] "Jakarta EE", Wikipedia

[11] "Spring Framework", Wikipedia

[12] "Hibernate (framework)", Wikipedia

[13] "Jakarta Persistence", JPA

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

[15] "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 © 2021 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.Arrays;

public class SimpleMainApp { ①
    public static final void main(String...args) { ② ③
        System.out.println("Hello " + Arrays.asList(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/javemain`) that match the Java package name of the class (`info.ejava.examples.app.build.javemain`)

```
|-- pom.xml ①
`-- src
  |-- main ②
  |   |-- java
  |   |   '-- info
  |   |       '-- ejava
  |   |           '-- examples
  |   |               '-- app
  |   |                   '-- build
  |   |                       '-- javemain
  |   |                           '-- 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>6.0.0-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 optional in this case. The default is to `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>6.0.0-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 11) that are compatible with our module

```

<properties>
    <java.target.version>11</java.target.version>
    <maven-compiler-plugin.version>3.8.1</maven-compiler-plugin.version>
    <maven-jar-plugin.version>3.1.2</maven-jar-plugin.version>
</properties>

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

```
$ mvn clean package
[INFO] Scanning for projects...
[INFO]
[INFO] -----< info.ejava.examples.app:java-app-example >-----
[INFO] Building App::Build::Java Main Example 6.0.0-SNAPSHOT
[INFO] -----[ jar ]-----
[INFO]
[INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ java-app-example ---
[INFO] Deleting .../java-app-example/target
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ java-app-example
---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 0 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.8.1:compile (default-compile) @ java-app-example
---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 1 source file to .../java-app-example/target/classes
[INFO]
[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @ java-
app-example ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 0 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.8.1:testCompile (default-testCompile) @ java-app-
example ---
[INFO] Changes detected - recompiling the module!
[INFO]
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ java-app-example ---
[INFO]
[INFO] --- maven-jar-plugin:3.1.2:jar (default-jar) @ java-app-example ---
[INFO] Building jar: .../java-app-example/target/java-app-example-6.0.0-SNAPSHOT.jar
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] Total time: 1.504 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-6.0.0-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 Archiver 3.4.0
Build-Jdk-Spec: 11
Main-Class: info.ejava.springboot.examples.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>
</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-surefire-plugin:2.12.4:test (default-test) @ java-app-example ---
[INFO]
[INFO] --- maven-jar-plugin:3.1.2:jar (default-jar) @ java-app-example ---
[INFO] Building jar: .../java-app-example/target/java-app-example-6.0.0-SNAPSHOT.jar
[INFO]
[INFO] --- maven-antrun-plugin:1.8:run (execute-jar) @ java-app-example -①
[WARNING] Parameter tasks is deprecated, use target instead
[INFO] Executing tasks

main:
      [java] Hello [Ant-supplied classpath, Command Line, args] ②
      [java] Hello [Ant-supplied executable jar, Command Line, args]
[INFO] Executed tasks
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 2.375 s
```

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 a class specified to 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 © 2021 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](#) pom that can be used as a parent pom for our Spring Boot modules.^[16] 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.

`springboot-starter-parent` inherits from a [spring-boot-dependencies](#) pom file that provides a definition of artifact versions but is less opinionated in 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.

[16] [Spring Boot and Maven, 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 of the modules.



"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>2.4.2</springboot.version>
</properties>
```



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>
  <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>
```

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 (thru 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>
```

29.2. Declare dependency on artifacts used

Enact 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>
    </dependency>
</dependencies>
```

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 final 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 thru 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
 - e.g., factory `@Bean` definitions
- `@EnableAutoConfiguration` - Allows Spring to perform auto-configuration 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](#). Lets extend our parent and local/leaf pom.xml files to enhance the standard JAR to be a Spring Boot executable JAR.

31.1.1. Define base use of [spring-boot-maven-plugin](#).

```
# Place this definition in an inherited parent pom
<build>
  <pluginManagement>
    <plugins>
      ...
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
        <version>${springboot.version}</version>
        <executions>
          <execution>
            <goals>
              <goal>repackage</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </pluginManagement>
</build>
```

- Addresses build issues constructing a JAR suitable for Spring Boot
- The above definition—when declared in the child pom—will cause this plugin to activate and modify the Maven-built JAR with constructs enabling Spring Boot to operate.

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. Declare spring-boot-maven-plugin in JAR module

Declare the `spring-boot-maven-plugin` in modules implementing the JAR hosting the Spring Boot Application.

```
# Place this declaration in the child/leaf pom building the JAR archive
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```

31.1.3. Build the JAR

```
$ mvn clean package

[INFO] Scanning for projects...
...
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ springboot-app-example -①
[INFO] Building jar: .../springboot-app-example/target/springboot-app-example-6.0.0-
SNAPSHOT.jar
[INFO]
[INFO] --- spring-boot-maven-plugin:2.4.2:repackage (default) @ springboot-app-example
-②
[INFO] Replacing main artifact with repackaged archive
[INFO] -----
[INFO] BUILD SUCCESS
```

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-6.0.0-SNAPSHOT.jar META-INF/MANIFEST.MF

Manifest-Version: 1.0
Archiver-Version: Plexus Archiver
Created-By: Apache Maven 3.6.1
Built-By: jim
Build-Jdk: 11.0.4
Main-Class: org.springframework.boot.loader.JarLauncher ①
Start-Class: info.ejava.examples.app.build.springboot.SpringBootApp ②
Spring-Boot-Version: 2.4.2
Spring-Boot-Classes: BOOT-INF/classes/
Spring-Boot-Lib: BOOT-INF/lib/
```

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 our earlier standard JAR.

```
springboot-app-example$ ls -lh ../*target/*.jar
-rw-r--r-- 1 jim staff 3.4K Nov 30 09:04 ../java-app-example/target/java-app-
example-6.0.0-SNAPSHOT.jar ①
-rw-r--r-- 1 jim staff 7.6M Dec  4 08:57 ../springboot-app-
example/target/springboot-app-example-6.0.0-SNAPSHOT.jar ②
```

① The earlier Java Main application is 3.4KB

② The Spring Boot JAR is 7.6MB

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 package 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-1.3.2.jar ②
...
BOOT-INF/lib/spring-context-5.1.9.RELEASE.jar
BOOT-INF/lib/spring-core-5.1.9.RELEASE.jar
BOOT-INF/lib/spring-expression-5.1.9.RELEASE.jar
BOOT-INF/lib/spring-jcl-5.1.9.RELEASE.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/ExecutableArchiveLauncher.class ①
org/springframework/boot/loader/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 also can use a WAR structure

- 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-6.0.0-SNAPSHOT.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-6.0.0-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. Lets 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.Arrays;

@Component ①
public class AppCommand implements CommandLineRunner {
    public void run(String... args) throws Exception {
        System.out.println("Component code says Hello " + Arrays.asList(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-6.0.0-SNAPSHOT.jar
```

Running SpringApplication ①

The Spring Boot logo is a stylized graphic composed of various symbols like slashes, parentheses, and dots. It features a large red circled number '2' at the top right. Below it, there's a complex arrangement of black symbols forming a shape that looks like a boot or a gear.

- 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. Directory Structure

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

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` or use the lower-level annotations
4. Place the application class in a Java package that is 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

Race Registration/Results

Assignment 0

copyright © 2021 jim stafford (jim.stafford@jhu.edu)

- Changes
 - 2021-09-02 - restored partB contents

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

As with most assignments, a set of starter projects is available in [assignment-starters/race-starters](#). 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
- extend either [spring-boot-starter-parent](#) or [ejava-build-parent](#)

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.

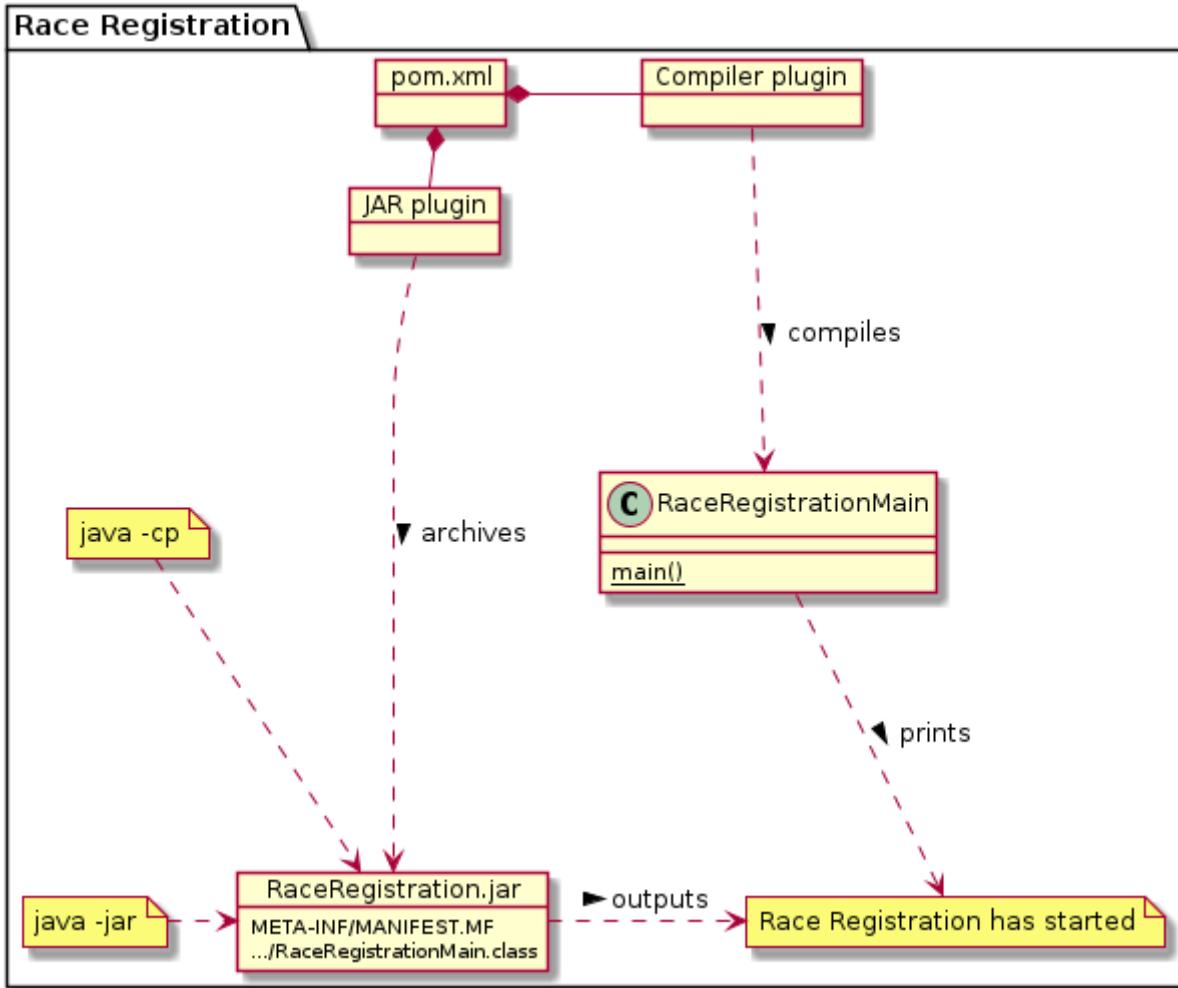


Figure 5. Pure Java Main Application

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 "Race Registration 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 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 it 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 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. 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.
3. A quick start project is available in `assignment-starters/race-starters/assignment0-race-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.

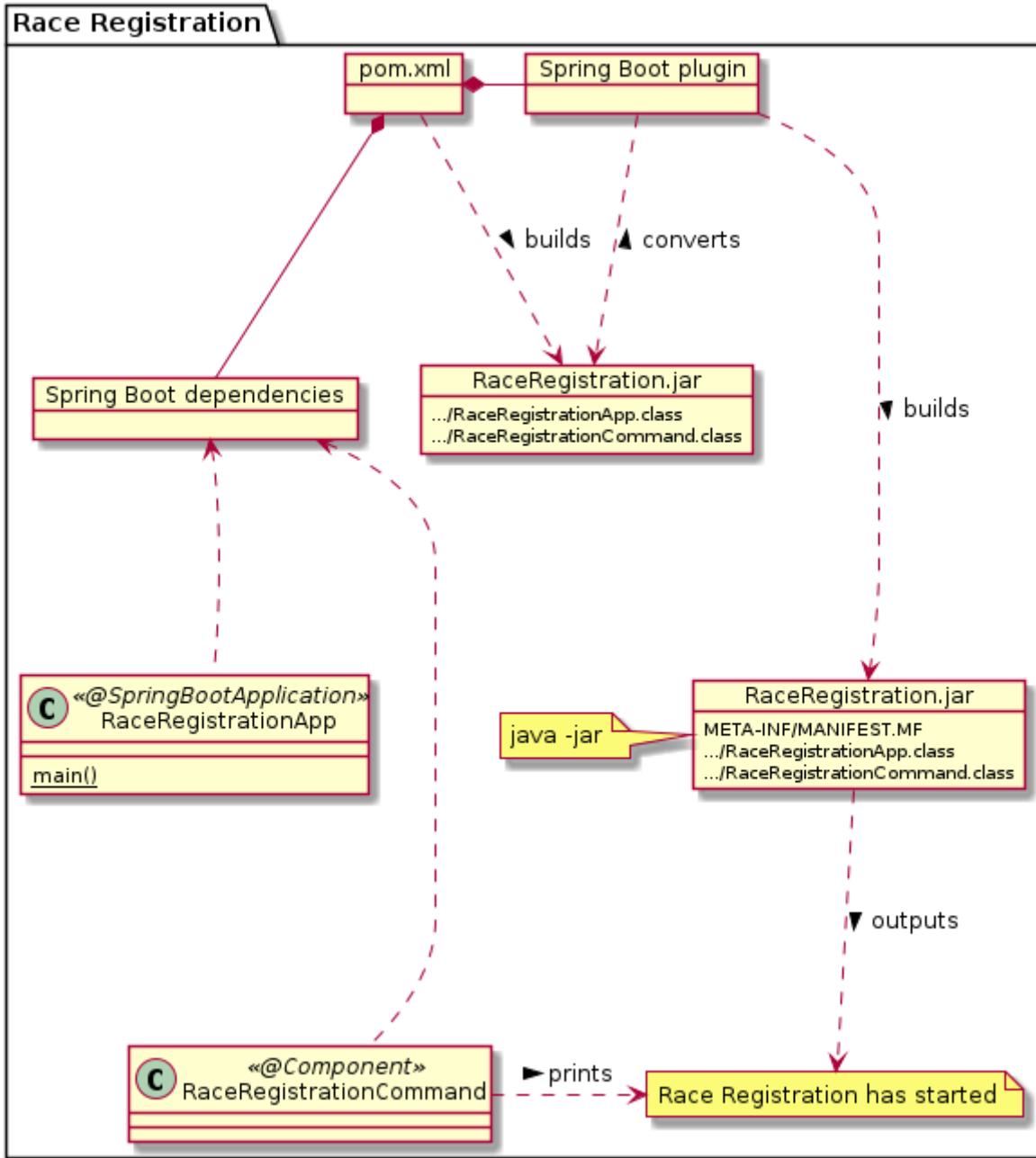


Figure 6. Spring Boot Application

37.3. Requirements

1. Create a Maven project that will host a Spring Boot Application
2. Supply a single Java class with a `main()` method that will bootstrap the Spring Boot Application
3. Supply a `@Component` that will be invoked when the application starts
 - a. have that `@Component` print a single "Race Registration 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` 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

- A quick start project is available in [assignment-starters/race-starters/assignment1-race-bootapp](#). Modify Maven groupId and Java package if used.

Bean Factory and Dependency Injection

copyright © 2021 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 sample `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>6.0.0-SNAPSHOT</version>
<artifactId>hello-service-api</artifactId>
<packaging>jar</packaging>
...
```

39.4. Hello Service Std 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>6.0.0-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.



Constructor injection makes required attributes marked final easier to set during testing

39.7. Hello Service Modules Complete

We are now done implementing our sample service interface and implementation—just build and install to make available to the application we will work on next.

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 6.0.0-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-6.0.0-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-6.0.0-SNAPSHOT.jar to .../.m2/repository/info/ejava/examples/app/hello-service-api/6.0.0-SNAPSHOT/hello-service-api-6.0.0-SNAPSHOT.jar
[INFO] Installing .../app-config/hello-service-api/pom.xml to .../.m2/repository/info/ejava/examples/app/hello-service-api/6.0.0-SNAPSHOT/hello-service-api-6.0.0-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 6.0.0-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-6.0.0-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-6.0.0-SNAPSHOT.jar to .../.m2/repository/info/ejava/examples/app/hello-service-stdout/6.0.0-SNAPSHOT/hello-service-stdout-6.0.0-SNAPSHOT.jar
[INFO] Installing .../app-config/hello-service-stdout/pom.xml to
.../.m2/repository/info/ejava/examples/app/hello-service-stdout/6.0.0-SNAPSHOT/hello-service-stdout-6.0.0-SNAPSHOT.pom
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 2.658 s
```

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:6.0.0-SNAPSHOT
[INFO] \- info.ejava.examples.app:hello-service-api:jar:6.0.0-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.



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 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 final 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 method the container will call when the container determines it needs a bean of a certain type and locates a `@Bean` annotated method with a return type of the required type.

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 final 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.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 final 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.jar | grep
applicationContext.xml
BOOT-INF/classes/context/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" xmlns:context=
"http://www.springframework.org/schema/context"
       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.jar
...
Xml @Bean says Hey World
```

Chapter 48. 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`

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 © 2021 jim stafford (jim.stafford@jhu.edu)

Chapter 49. 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");  
}
```

Lets say we don't want the value hard-coded or passed in as a command-line argument. Lets 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](#)

49.1. Goals

The student will learn:

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

49.2. 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 50. @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`

50.1. Value Not Found

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

```
2019-09-22 20:16:24.286  WARN 38915 --- [main]
s.c.a.AnnotationConfigApplicationContext :
Exception encountered during context initialization - cancelling refresh attempt:
org.springframework.beans.factory.BeanCreationException: Error creating bean with
name 'appCommand': Injection of autowired dependencies failed; nested exception
is java.lang.IllegalArgumentException: Could not resolve placeholder
'app.audience' in value "${app.audience}"
```

50.2. 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.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

50.3. 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.jar
...
Application @Bean says Hey Default World
```

Property Defined

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

Chapter 51. 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

An alternative to using `field` injection is to change it to `constructor` injection.

```
@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 52. Property Types

52.1. 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.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
```

52.2. 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 prased sequence with duplicates injected into Set retained only unique values
- ③ parsed sequence with duplicates injected into Array maintained duplicates

52.3. Custom Delimiters (using Spring EL)

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) `#{} 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!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

52.4. 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

52.5. 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

52.6. 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

52.7. 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.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 53. 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 © 2021 jim stafford (jim.stafford@jhu.edu)

Chapter 54. 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 setup ways to specify entire collection of property values through files.

54.1. 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

54.2. 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 straight `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

Chapter 55. 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.

55.1. 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.jar | \
    egrep 'BOOT-INF/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
```

55.2. 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.jar  
...  
Application @Bean says Hey application.properties value
```

can also be completed with

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

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

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

55.3. 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-*-.jar \  
  --spring.config.name=property_source  
...  
Application @Bean says Hey Property Source value
```

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

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

55.4. 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-*-.jar \  
  --spring.config.name="property_source,alternate_source"  
...  
Application @Bean says Hey alternate source property file
```

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

Chapter 56. @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.jar  
...  
Application @Bean says Hey Property Source value
```

Chapter 57. 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.

57.1. 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-*-.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

57.2. 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-*-.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

57.3. 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-*-.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-*-.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

57.4. 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.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 58. 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
```

58.1. 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
```

58.2. 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 the 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

58.3. 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.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`

58.4. 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-*-SNAPSHOT.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`

58.5. 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-*-SNAPSHOT.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`

58.6. 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-*-SNAPSHOT.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 59. 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

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 © 2021 jim stafford (jim.stafford@jhu.edu)

Chapter 60. 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 individual configuration properties entirely 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.

60.1. 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

60.2. 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 `@ConstructorBinding`
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 61. 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
```

61.1. 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

61.2. 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

61.3. 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

61.4. Registering the `@ConfigurationProperties` class

We currently have a similar problem that we had when we implemented our first `@Configuration` and `@Component` classes—the bean is 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:

```
src
`-- main
  |-- java
  |  |-- info
  |  |  |-- ejava
  |  |  |  |-- examples
  |  |  |  |  |-- app
  |  |  |  |  |  |-- config
  |  |  |  |  |  |  |-- configproperties
  |  |  |  |  |  |  |  |-- AppCommand.java
  |  |  |  |  |  |  |  |-- ConfigurationPropertiesApp.java
  |  |  |  |  |  |  |  |-- properties
  |  |  |  |  |  |  |  |  |-- CarProperties.java
  |-- resources
    |-- application.properties
```

61.4.1. 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.

```
@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

61.4.2. way 2 - Explicitly Register Class

Explicitly register the class using `@EnableConfigurationProperties` annotation on a `@Configuration` class (such as the `@SpringBootApplication` class)

```
@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

61.4.3. way 3 - Enable Package Scanning

Enable package scanning for `@ConfigurationProperties` classes with the `@ConfigurationPropertiesScan` annotation

```
@SpringBootApplication  
@ConfigurationPropertiesScan ①  
public class ConfigurationPropertiesApp {
```

① allows a generalized scan to be defined that is separate for configurations

- benefits: easy to add more configuration classes without changing application
- drawbacks: generalized scan may accidentally pick up an unwanted configuration

61.4.4. way 4 - Use `@Bean` factory

Create a `@Bean` factory method in a `@Configuration` class for the type .

```
@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 for this example, I am going to use `@ConfigurationPropertiesScan ("way3")` and drop

multiple `@ConfigurationProperties` classes into the same classpath and have them automatically scanned for.

61.5. 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.jar
...
carProperties=CarProperties{name='Suburban'}
```

Chapter 62. Metadata

IDEs have support for linking Java properties to their `@ConfigurationProperty` class information.

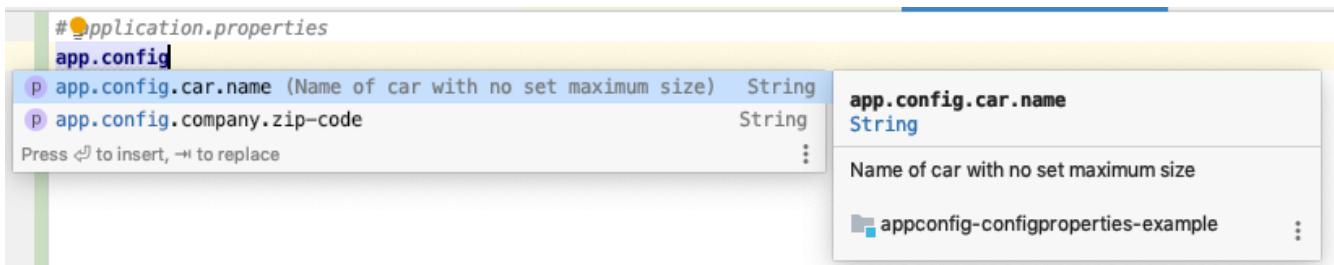


Figure 7. IDE Configuration Property Support

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!)

62.1. 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.

62.2. 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.

62.3. 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

62.4. 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
```

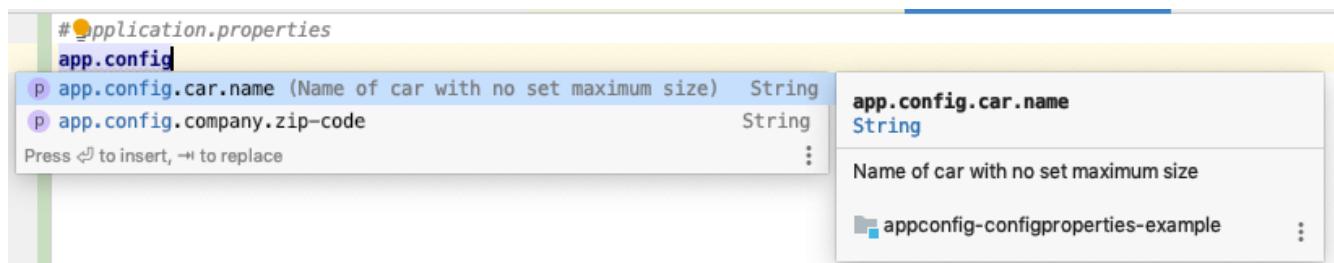
Produced Metadata File in `target/classes` Tree

```
target/classes/META-INF/
`-- spring-configuration-metadata.json
```

```
{  
  "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": []  
}
```

62.5. 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 63. 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.

The only requirement Spring places on us is to add a `@ConstructorBinding` annotation to the class or constructor method when using this approach.

Constructor Binding Example

```
...
import org.springframework.boot.context.properties.ConstructorBinding;

@ConfigurationProperties("app.config.boat")
public class BoatProperties {
    private final String name; ③

    @ConstructorBinding ②
    public BoatProperties(String name) {
        this.name = name;
    }
    //no setter method(s) 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 the class or constructor with `@ConstructorBinding`

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.

63.1. 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-*-.jar  
...  
boatProperties=BoatProperties{name='Maxum'}
```

63.2. 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-*-.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 64. 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](#)

64.1. 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 [@NotNull](#) constraint to report an error if the property is ever null.

```
...
import org.springframework.validation.annotation.Validated;
import javax.validation.constraints.NotNull;

@ConfigurationProperties("app.config.boat")
@Validated ①
public class BoatProperties {
    @NotNull 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 `@NotNull` annotation tells the validator this field is not allowed to be null



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.

64.2. 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-*-.jar
*****
APPLICATION FAILED TO START
*****
Description:

Binding to target org.springframework.boot.context.properties.bind.BindException:
Failed to bind properties under 'app.config.boat' to
info.ejava.examples.app.config.configproperties.properties.BoatProperties failed:

Property: app.config.boat.name
Value: null
Reason: must not be null

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 65. 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 {
    @NotNull
    private final String name;

    @ConstructorBinding
    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

65.1. 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")
@ConstructorBinding
@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`

65.2. 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
```

i 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();  
}
```

65.3. 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>
```

65.4. 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.jar
...
boatProperties=BoatProperties{name='Maxum'}
=====
companyProperties=CompanyProperties(name=Acme)
```

There is a Spring [@ConstructorBinding](#) issue that prevents property metadata from being automatically generated. This is due to a [Lombok](#) issue where usable argument names are not provided in the generated constructor. The only workaround at this time if you want metadata generated for [@ConstructorBinding](#) with Lombok is to provide a custom constructor supplying the valid names. The IDE is very good at generating these for you until that issue is corrected.



```
@ConfigurationProperties("app.config.company")
@ConstructorBinding
@Data
@Validated
public class CompanyProperties {
    @NotNull
    private final String name;

    //https://github.com/spring-projects/spring-boot/issues/18730
    //https://github.com/rzwitserloot/lombok/issues/2275
    public CompanyProperties(String name) {
        this.name = name;
    }
}
```

Lombok ConstructorBinding Issue Listed as Closed



Since providing the warning above, the version of Lombok has advanced in class ([1.18.20](#)), issue closed, and may have been resolved. Confirmation needed.

With the exception of the property metadata issue just mentioned, 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 66. 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

66.1. 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")
@ConstructorBinding
@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;
}
```

66.2. 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`

66.3. 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.jar
...
businessProperties=BusinessProperties(name=Acme, streetAddress=100 Suburban Dr,
city=Newark, state=DE, zipCode=19711, notes=null)
```

Chapter 67. 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

67.1. 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  
@ConstructorBinding  
public class AddressProperties {  
    private final String street;  
    @NotNull  
    private final String city;  
    @NotNull  
    private final String state;  
    @NotNull  
    private final String zip;  
}
```



In this specific case we are using a read-only JavaBean and need to supply the `@ConstructorBinding` annotation.

67.2. 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")
@ConstructorBinding
@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.

67.3. 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-*-.jar
...
corporationProperties=CorporationProperties(name=Acme,
    address=AddressProperties(street=null, city>Newark, state>DE, zip>19711))
```

Chapter 68. 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")
@ConstructorBinding
@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`

68.1. 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
```

68.2. 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.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 69. 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")
@ConstructorBinding
@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`

69.1. 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.jar  
...  
UserProperties(name=jim, home=/Users/jim, timezone=America/New_York)  
user.home=/Users/jim
```

Chapter 70. @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

70.1. @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

70.2. `@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.

70.3. 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.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.

70.4. Injection Matching

Spring had two `@Bean` factory methods to chose 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

70.5. 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.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
```

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

70.6. 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.

70.7. 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

70.8. 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

```

@.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

70.9. 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

70.10. 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 71. 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 © 2021 jim stafford (jim.stafford@jhu.edu)

Chapter 72. 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, 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.

72.1. Goals

The student will learn to:

- Enable/disable `@Configuration` classes and `@Bean` factories based on condition(s) at startup
- Create an Auto-configuration ("Starter") module that establishes necessary dependencies and conditionally supplies beans
- Resolve conflicts between alternate configurations
- Locate environment and condition details to debug Auto-configuration issues

72.2. Objectives

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

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 a `spring.factories` 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 73. 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 final void main(String...args) {
        SpringApplication.run(SelfConfiguredApp.class, args);
    }

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

73.1. 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)
public class AConfigurationClass {
    @Bean
    public Hello hello() {
        return new StdOutHello("...");
    }
}
```

 `@Configuration` classes are commonly annotated with the `proxyMethods=false` attribute that tells Spring it need not 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 74. Conditional Configuration

We can make `@Bean` factory methods 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 equaling 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 final 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 equaling the value `true`

74.1. 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-example-*-.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.

74.2. Property Value Condition Not Satisfied

The following is an example of the property being 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-example-*-.SNAPSHOT.jar ①
...
*****
APPLICATION FAILED TO START
*****
```

Description:

Parameter 0 of constructor in `info.ejava.springboot.examples.app.AppCommand` required a bean of type
'`info.ejava.examples.app.hello.Hello`' that could not be found.

The following candidates were found but could not be injected: ②

- Bean method 'quietHello' in 'StarterConfiguredApp' not loaded because
`@ConditionalOnProperty (hello.quiet=true)` did not find property 'quiet'

Action:

Consider revisiting the entries above or 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 75. Two Primary Configuration Phases

Configuration processing within Spring Boot is broken 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 76. Auto-Configuration

An Auto-configuration class is technically no different than 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 `META-INF/spring.factories` descriptor. 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 that is 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.

76.1. 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 {
    @NotNull
    private String greeting = "HelloProperties default greeting says Hola!"; ①
}
```

① Value used if user-configuration does not specify a property value

76.2. Locating Auto Configuration Classes

Auto-configuration class(es) are registered with an entry within the `META-INF/spring.factories` file of the Auto-configuration class` JAR. This module is typically called a "starter".

Auto-configuration Module JAR

```
$ jar tf target/hello-starter-*-SNAPSHOT.jar | egrep -v '/$|maven|MANIFEST.MF'
META-INF/spring.factories ①
META-INF/spring-configuration-metadata.json ②
info/ejava/examples/app/hello/HelloAutoConfiguration.class
info/ejava/examples/app/hello>HelloProperties.class
```

① "starter" dependency JAR supplies `META-INF/spring.factories`

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.

76.3. META-INF/spring.factories Metadata File

The Auto-configuration classes are 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 as I will show later.

```
# src/main/resources/META-INF/spring.factories
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
    info.ejava.examples.app.hello>HelloAutoConfiguration ①
```

① Auto-configuration class metadata registration

76.4. Example Starter Module Source Tree

Our configuration and properties class—along with the `spring.factories` file get placed in a separate module source tree.

Example Starter Module Structure

```
pom.xml
src
`-- main
    |-- java
    |   |-- info
    |   |   |-- ejava
    |   |   |   |-- examples
    |   |   |   |   |-- app
    |   |   |   |   |   |-- hello
    |   |   |   |   |   |   |-- HelloAutoConfiguration.java
    |   |   |   |   |   |   |-- HelloProperties.java
    |-- resources
        |-- META-INF
            |-- spring.factories
```

76.5. Example Starter Module pom.xml

The module is commonly termed a `starter` and will have dependencies on

- `spring-boot-starter`
- 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

76.6. Example Starter Implementation Dependencies

The rest of the dependencies have nothing specific to do with Auto-configuration or starter modules and are there to support the module implementation.

Example Starter pom.xml Implementation Dependencies

```
<dependency> ①
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <scope>provided</scope>
</dependency>
<dependency> ①
  <groupId>javax.validation</groupId>
  <artifactId>validation-api</artifactId>
</dependency>

<!-- creates a JSON metadata file describing @ConfigurationProperties -->
<dependency> ①
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-configuration-processor</artifactId>
  <optional>true</optional>
</dependency>
</dependencies>
```

- ① these dependencies are part of optional implementation detail having nothing to do with Auto-configuration topic

76.7. Application Starter Dependency

The application module declares dependency on the starter module containing 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.

76.8. 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:6.0.0-SNAPSHOT:compile
[INFO] |  +- info.ejava.examples.app:hello-service-api:jar:6.0.0-SNAPSHOT:compile
[INFO] |  +- info.ejava.examples.app:hello-service-stdout:jar:6.0.0-SNAPSHOT:compile
[INFO] |  +- org.projectlombok:lombok:jar:1.18.10:provided
[INFO] |  \- org.springframework.boot:spring-boot-starter-validation:jar:2.2.1.RELEASE:compile
[INFO] |     +- jakarta.validation:jakarta.validation-api:jar:2.0.1:compile
[INFO] |     +- org.apache.tomcat.embed:tomcat-embed-el:jar:9.0.27:compile
[INFO] |     \- org.hibernate.validator:hibernate-validator:jar:6.0.18.Final:compile
[INFO] |         +- org.jboss.logging:jboss-logging:jar:3.4.1.Final:compile
[INFO] |         \- com.fasterxml:classmate:jar:1.5.1:compile
```

Chapter 77. 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;

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

① lombok is being used to provide the constructor injection

77.1. Review: Unconditional Auto-Configuration Class

This starter module is providing a `@Bean` factory to construct an implementation of `Hello`.

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`

77.2. Review: Starter Module Default

The starter module defines an Auto-configuration class that instantiates a `StdOutHello` implementation configured by a `HelloProperties` class.

```
@ConfigurationProperties("hello")
@Data
@Validated
public class HelloProperties {
    @NotNull
    private String greeting = "HelloProperties default greeting says Hola!"; ①
}
```

① `hello.greeting` default defined in `@ConfigurationProperties` class of starter module

77.3. Produced Default Starter Greeting

This produces the default greeting

Example Application Execution without Satisfying Property Condition

```
$ java -jar target/appconfig-autoconfig-example-*-.jar
...
HelloProperties default greeting says Hola! World
```

77.4. 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-example-*-.jar
...
application.properties Says - Hey World ①
```

① auto-configured implementation using user-defined property

Chapter 78. Auto-Configuration Conflict

78.1. 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)");
    }
}
```

78.2. Potential Conflict

We also saw how to define `@Bean` factory in an Auto-configuration class packaged in a 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-example-*-SNAPSHOT.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]
```

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 chose from

78.3. `@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 starter modules. However, the `@ConditionalOnMissingBean` and its sibling `@ConditionalOnBean` are special and meant to be used with Auto-configuration classes in the starter 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` factory 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

78.4. 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-example-*-SNAPSHOT.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-example-*-SNAPSHOT.jar  
...  
application.properties Says - Hey World
```

Chapter 79. 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;

@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`

79.1. 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 Auto-configuration property within the `META-INF/spring.factories` file. We do this by listing the full classnames of each Auto-configuration class, separated by comma(s).

`hello-starter` `spring.factories`

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
    info.ejava.examples.app.hello.HelloAutoConfiguration, \ ①
    info.ejava.examples.app.hello.HelloResourceAutoConfiguration
```

① comma separated

79.2. 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

```
$ echo hello.greeting: hello.properties exists says hello World > hello.properties
$ cat hello.properties
hello.greeting: hello.properties exists says hello World

$ java -jar target/appconfig-autoconfig-example-*-SNAPSHOT.jar
...
hello.properties exists says hello World
```

- when property file is not present
 - `@Bean` factory from `HelloAutoConfiguration` used since neither property or resource-based conditions satisfied

Resource Condition Not Satisfied

```
$ rm hello.properties
$ java -jar target/appconfig-autoconfig-example-*-SNAPSHOT.jar
...
application.properties Says - Hey World
```

Chapter 80. @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-example-*-.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;

@ConditionalOnResource(resources = "file:./hello.properties")
@AutoConfigureBefore(HelloAutoConfiguration.class)
public class HelloResourceAutoConfiguration {
    @Bean
    @Primary //chosen when there is a conflict
    public Hello resourceHello() {
        return new StdOutHello("hello.properties exists says hello");
    }
}
```

80.1. @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 thru @Primary

```
$ cat hello.properties
hello.greeting: hello.properties exists says hello World
$ java -jar target/appconfig-autoconfig-example-*-.SNAPSHOT.jar --hello.quiet=true ①
...
hello.properties exists says hello World
```

① `@Primary` condition satisfied overrides application `@Bean` condition

Chapter 81. 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 prior to 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.

81.1. 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 82. 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 83. 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.

83.1. 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.

83.2. Conditions Evaluation Report Example

Conditions Evaluation Report Snippet

```
$ java -jar target/appconfig-autoconfig-example-*-.SNAPSHOT.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:
  - @ConditionalOnBean (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)

StarterConfiguredApp#quietHello:
  Did not match:
    - @ConditionalOnProperty (hello.quiet=true) did not find property 'quiet'
(OnPropertyCondition)
```

① Positive matches show which conditionals are activated and why

② Negative matches show which conditionals are not activated and why

83.3. 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

83.4. 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>
```

83.5. 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-example-*-SNAPSHOT.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": []
},
...
}
}
}
```

83.6. 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-example-*-SNAPSHOT.jar \
--management.endpoints.web.exposure.include=conditions,env
```

83.7. 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
    }
  }
}
```

83.8. 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"  
        }  
    }  
,  
...  
]
```

83.9. 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"  
},  
...  
]
```

83.10. 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-example-*-.jar \  
--management.endpoints.web.exposure.include="*" ①
```

① double quotes ("") being used to escape `*` special character on command line

Chapter 84. 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 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 © 2021 jim stafford (jim.stafford@jhu.edu)

Chapter 85. Introduction

85.1. 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?

85.2. 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).

85.3. 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

85.4. 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 86. 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 top-tier 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 87. 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:6.0.0-SNAPSHOT
[INFO] \- org.springframework.boot:spring-boot-starter:jar:2.2.1.RELEASE:compile
[INFO]   +- org.springframework.boot:spring-boot:jar:2.2.1.RELEASE:compile
...
[INFO]   +- org.springframework.boot:spring-boot-
autoconfigure:jar:2.2.1.RELEASE:compile
[INFO]   +- org.springframework.boot:spring-boot-starter-
logging:jar:2.2.1.RELEASE:compile ①
[INFO]     |  +- ch.qos.logback:logback-classic:jar:1.2.3:compile
[INFO]     |  +- ch.qos.logback:logback-core:jar:1.2.3:compile
[INFO]     |  \- org.slf4j:slf4j-api:jar:1.7.29:compile
[INFO]     |  +- org.apache.logging.log4j:log4j-to-slf4j:jar:2.12.1:compile
[INFO]     |  \- org.apache.logging.log4j:log4j-api:jar:2.12.1:compile
[INFO]     \- org.slf4j:jul-to-slf4j:jar:1.7.29:compile
...
[INFO]   +- org.springframework:spring-core:jar:5.2.1.RELEASE:compile
[INFO]     \- org.springframework:spring-jcl:jar:5.2.1.RELEASE:compile
...
```

① dependency on [spring-boot-starter](#) brings in [spring-boot-starter-logging](#)

87.1. 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.

87.2. 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 88. 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.

88.1. 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

88.2. 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.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`

88.3. 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".

88.4. Getting a Logger

Logging frameworks make use of the fundamental design idiom—separate interface from implementation. We want our calling to 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 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.

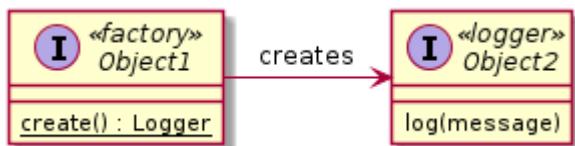


Figure 8. Logging Framework Primary Stereotypes

- Factory creates Logger

Lets take a look at several ways to obtain a Logger using different APIs and techniques.

88.5. 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.

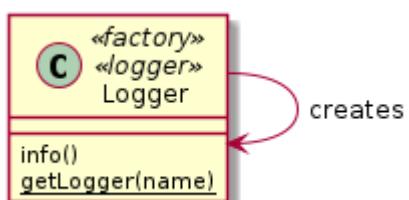


Figure 9. Java Util Logger (JUL) Logger

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.

88.6. 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.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.

88.7. 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.

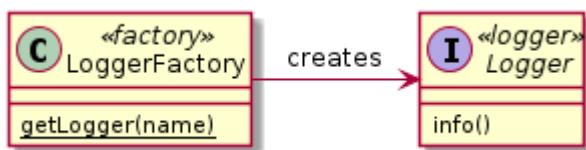


Figure 10. SLF4J LoggerFactory and Logger

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

i 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 are called from. The `String` form is also available but the `Class` form helps encourage and simplify following a common best practice.

88.8. SLF4J Example Output

SLF4J Example Output

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT.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

88.9. Lombok SLF4J Declaration Example

Naming loggers after the fully qualified classname 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

88.10. 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.jar \
--spring.profiles.active=factory
...
20:40:55,155 INFO  info.ejava.examples.app.config.logging.factory.LombokDeclaredLogger
- lombok declared SLF4J logger message
...
```

88.11. 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 89. 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.

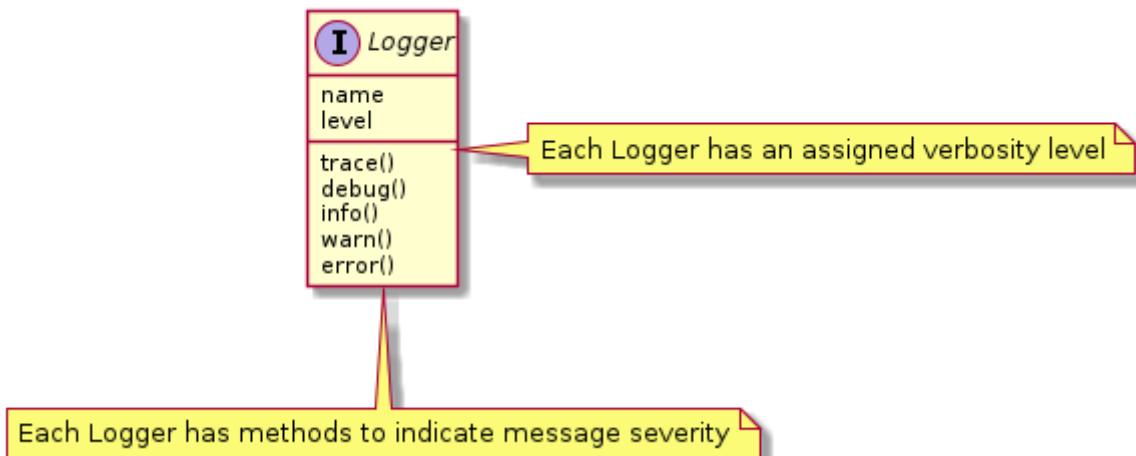


Figure 11. Logging Level

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.

89.1. 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 try to live by:

TRACE

Detailed audits events and verbose state of processing

- example: Detailed state at a point in the code. The SQL used in a query.

DEBUG

Audit events and state giving insight of actions performed

- example: Beginning/ending a decision branch in the code 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 from 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

89.2. 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.

89.3. 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

89.4. 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.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

89.5. 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.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

89.6. 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.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

89.7. 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.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 `WARN`

89.8. 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.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 90. 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

90.1. 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

90.2. 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.

90.2.1. 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

90.2.2. 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()`

90.3. 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

90.4. 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 any noticeable 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 any noticeable penalty

Disabled Logger Relative Results

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT.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)

90.5. 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.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 91. 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

91.1. 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.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

91.2. 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 92. 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%n
```

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-*-.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
```

92.1. 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](#)

92.2. 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](#).

```
java -jar target/appconfig-logging-example-*-.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 12. Default Spring Boot Console Log Pattern Coloring

92.3. 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.

92.4. 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.

92.5. 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.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

92.6. 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.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

92.7. 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

92.8. 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>logging.pattern.level</code>	<code>output</code>	<code>comment</code>
<code>[%.-2level]</code>	<code>[IN] [WA] [ER]</code>	value takes no more than 2 characters, starting from the left
<code>[%.2level]</code>	<code>[FO] [RN] [OR]</code>	value takes no more than 2 characters, starting from the right

92.9. 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

92.10. 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.

92.11. Example Override Output

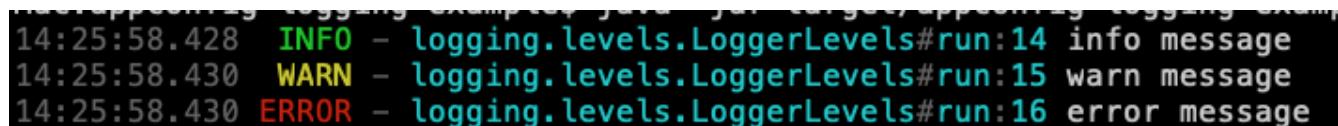
We can activate the profile and demonstrate the modified format using the following command.

Example Console Pattern Override Output

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT.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.



A terminal window displaying the same log output as above, but with color applied to the log levels. The word 'INFO' is in green, 'WARN' is in yellow, and 'ERROR' is in red. The log entries are:
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 13. Example Console Pattern Override Coloring

92.12. Layout Fields

Please see the [Logback Layouts Documentation](#) for a detailed list of conversion words and how to optionally format them.

Chapter 93. 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

93.1. 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.

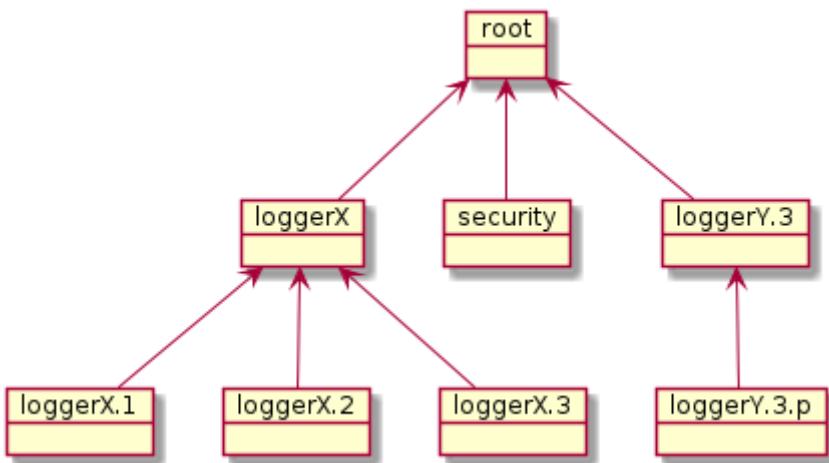


Figure 14. Example Logger Tree

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

93.2. 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).

93.3. 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

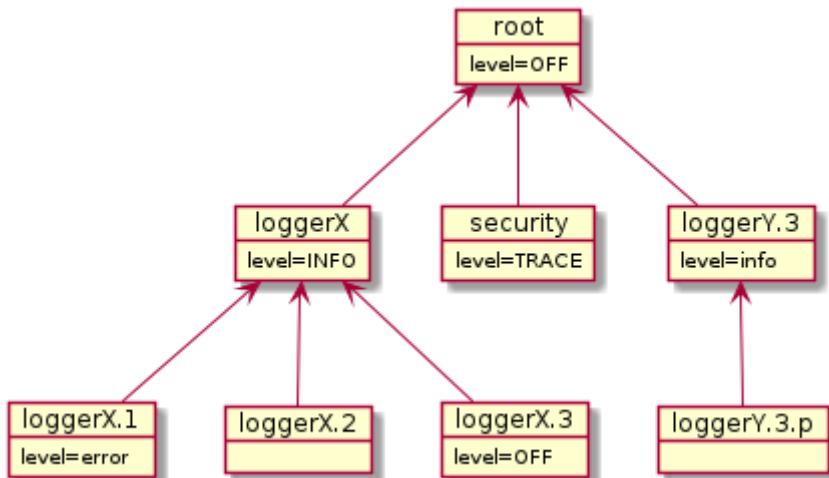


Figure 15. Logger Level Inheritance

93.4. 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

93.5. Example Logger Threshold Level Properties

These thresholds can be expressed in a property file.

```
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
```

93.6. 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.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 94. 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

94.1. Logger has N Appenders

Each of the loggers in our tree has the chance to have 0..N appenders.



Figure 16. Logger / Appender Relationship

94.2. 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

94.3. 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">
    ...
</configuration>
```

94.4. Retain Spring Boot Defaults

We will loose most/all of the 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 debug="false">
    <!-- bring in Spring Boot defaults for Logback -->
    <include resource="org/springframework/boot/logging/logback/defaults.xml"/>
    ...
</configuration>
```

94.5. Appender Configuration

Our example tree has three (3) appenders total. Each adds a literal string prefix so we know which appender is being called.

```
<!-- 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>
        <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



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.

94.6. 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.

```

<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`

94.7. 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`.

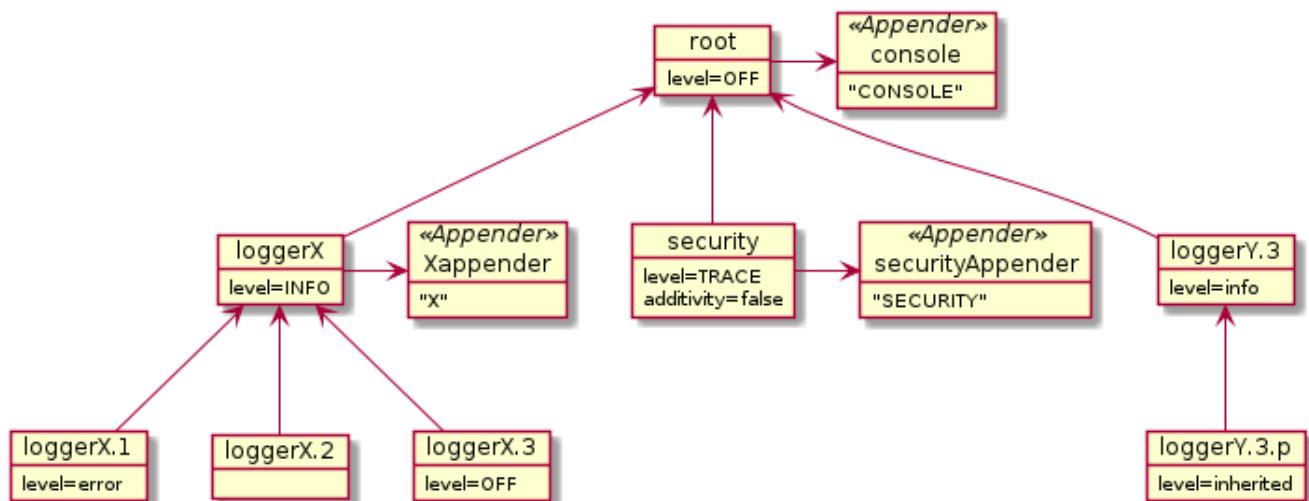


Figure 17. Example Logger Tree with Appenders

94.8. 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

94.9. Logger Inheritance Tree Output

```
$ java -jar target/appconfig-logging-example-*-.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 95. 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.

95.1. 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

95.2. 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

95.3. 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.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.

Chapter 96. 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!!!")` ^[17]



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.

96.1. 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.

96.2. 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

96.3. 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>
```

96.4. 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.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

[17] "what are markers in java logging frameworks and what is a reason to use them", Stack Overflow, 2019

Chapter 97. 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-*-.jar \
--spring.profiles.active=levels \
--logging.file.name="mylog.log" ① ②
```

① adding this property adds file logging to default configuration

② this expressed logfile will be written to **mylog.log** in current directory

97.1. 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.

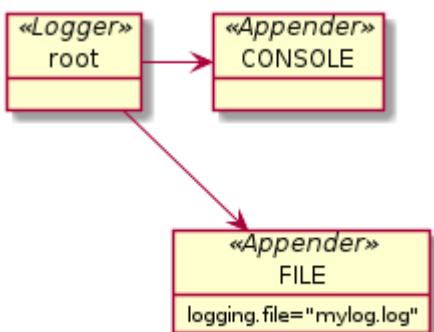


Figure 18. root Logger Appenders

97.2. 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

97.3. 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](#).

```
<appender name="FILE"
    class="ch.qos.logback.core.rolling.RollingFileAppender">①
    <encoder>
        <pattern>${FILE_LOG_PATTERN}</pattern>
    </encoder>
    <file>${LOG_FILE}</file>
    <rollingPolicy class="ch.qos.logback.core.rolling.SizeAndTimeBasedRollingPolicy">
        ②
            <cleanHistoryOnStart>${LOG_FILE_CLEAN_HISTORY_ON_START:-false}</cleanHistoryOnStart>
            <fileNamePattern>${ROLLING_FILE_NAME_PATTERN:-${LOG_FILE}.%d{yyyy-MM-dd}.%i.gz}</fileNamePattern>
            <maxFileSize>${LOG_FILE_MAX_SIZE:-10MB}</maxFileSize>
            <maxHistory>${LOG_FILE_MAX_HISTORY:-7}</maxHistory>
            <totalSizeCap>${LOG_FILE_TOTAL_SIZE_CAP:-0}</totalSizeCap>
        </rollingPolicy>
    </appender>
```

① performs file rollover functionality based on configured policy

② specifies policy and policy configuration to use

97.4. 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

97.5. 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
- define an option to process quotas at startup in addition to file changeover for short running applications

97.6. FILE Appender Properties

name	description	default
logging.file.path	full or relative path of directory written to — ignored when <code>logging.file.name</code> provided	.
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.file.max-size	maximum size of log before changeover — must be less than <code>total-size-cap</code>	10MB
logging.file.max-history	maximum number of historical files to retain when changing over because of date criteria	7
logging.file.total-size-cap	maximum amount of total space to consume — must be greater than <code>max-size</code>	(no limit)
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>



If file logger property value is invalid, the application will run without the FILE appender activated.

97.7. 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.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

97.8. 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.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

97.9. logging.file.max-size Trigger

One trigger for changing over to the next file is `logging.file.max-size`. The condition is satisfied when the current logfile reaches this value. The default is 10MB.

The following example changes that to 500 Bytes. Once each instance of `logging.file.name` reached the `logging.file.max-size`, it is compressed and moved to a filename with the pattern from `logging.pattern.rolling-file-name`.

logging.file.max-size Example

```
java -jar target/appconfig-logging-example-*-SNAPSHOT.jar \
--spring.profiles.active=rollover \
--logging.file.name=target/logs/mylog.log \
--logging.file.max-size=500B ①
...
$ ls -ltr target/logs

total 40
-rw-r--r-- 1 jim  staff  154 Mar 29 16:00 mylog.log.2020-03-29.0.gz
-rw-r--r-- 1 jim  staff  153 Mar 29 16:00 mylog.log.2020-03-29.1.gz
-rw-r--r-- 1 jim  staff  156 Mar 29 16:00 mylog.log.2020-03-29.2.gz
-rw-r--r-- 1 jim  staff  156 Mar 29 16:00 mylog.log.2020-03-29.3.gz ②
-rw-r--r-- 1 jim  staff  240 Mar 29 16:00 mylog.log ①
```

① `logging.file.max-size` limits the size of the current logfile

② historical logfiles renamed according to `logging.pattern.rolling-file-name` pattern

97.10. 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 and the FILE appender will be disabled if not specified
- the timestamp pattern directly impacts when the file changeover will occur when we are still below the `logging.file.max-size`. In that case—the changeover occurs 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.file.max-size` triggers the changeover and we already have a historical name with the same timestamp.
- the number of historical files is throttled using `logging.file.max-history` only when index is used and not when file changeover is due to `logging.file.max-size`
- the historical file will be compressed if `gz` is specified as the suffix

97.11. 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.jar \
--spring.profiles.active=rollover \
--logging.file.name=target/logs/mylog.log \
--logging.file.max-size=500 \
--logging.pattern.rolling-file-name='${logging.file.name}.%d{yyyy-MM-dd \
-HH:mm:ss}.%i'.log ①
...
$ ls -ltr target/logs

total 64
-rw-r--r-- 1 jim staff 79 Mar 29 17:50 mylog.log.2020-03-29-17:50:22.0.log ①
-rw-r--r-- 1 jim staff 79 Mar 29 17:50 mylog.log.2020-03-29-17:50:23.0.log
-rw-r--r-- 1 jim staff 79 Mar 29 17:50 mylog.log.2020-03-29-17:50:24.0.log
-rw-r--r-- 1 jim staff 79 Mar 29 17:50 mylog.log.2020-03-29-17:50:25.0.log
-rw-r--r-- 1 jim staff 79 Mar 29 17:50 mylog.log.2020-03-29-17:50:26.0.log
-rw-r--r-- 1 jim staff 80 Mar 29 17:50 mylog.log.2020-03-29-17:50:27.0.log
-rw-r--r-- 1 jim staff 80 Mar 29 17:50 mylog.log.2020-03-29-17:50:28.0.log
-rw-r--r-- 1 jim staff 80 Mar 29 17:50 mylog.log

$ file target/logs/mylog.log.2020-03-29-17\:50\:28.0.log ②
target/logs/mylog.log.2020-03-29-17:50:28.0.log: ASCII text
```

① `logging.pattern.rolling-file-name` pattern triggers changeover at the seconds boundary

② historical logfiles are left uncompressed because of name suffix specified



Using a date pattern to include minutes and seconds is just for demonstration and learning purposes. Most patterns would be daily.

97.12. 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.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'.gz ①
...
$ ls -ltr target/logs

total 64
-rw-r--r-- 1 jim staff 97 Mar 29 16:26 mylog.log.2020-03-29-16:26:11.0.gz ①
-rw-r--r-- 1 jim staff 97 Mar 29 16:26 mylog.log.2020-03-29-16:26:12.0.gz
-rw-r--r-- 1 jim staff 97 Mar 29 16:26 mylog.log.2020-03-29-16:26:13.0.gz
-rw-r--r-- 1 jim staff 97 Mar 29 16:26 mylog.log.2020-03-29-16:26:14.0.gz
-rw-r--r-- 1 jim staff 97 Mar 29 16:26 mylog.log.2020-03-29-16:26:15.0.gz
-rw-r--r-- 1 jim staff 97 Mar 29 16:26 mylog.log.2020-03-29-16:26:16.0.gz
-rw-r--r-- 1 jim staff 79 Mar 29 16:26 mylog.log
-rw-r--r-- 1 jim staff 97 Mar 29 16:26 mylog.log.2020-03-29-16:26:17.0.gz

$ file target/logs/mylog.log.2020-03-29-16\:26\:16.0.gz
target/logs/mylog.log.2020-03-29-16:26:16.0.gz: \
gzip compressed data, from FAT filesystem (MS-DOS, OS/2, NT), original size 79
```

① historical logfiles are compressed when pattern uses a `.gz` suffix

97.13. `logging.file.max-history` Example

`logging.file.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.file.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.jar \
--spring.profiles.active=rollover \
--logging.file.name=target/logs/mylog.log \
--logging.file.max-size=250 \
--logging.pattern.rolling-file-name='${logging.file.name}.%d{yyyy-MM-dd}
-HH:mm:ss}.%i'.log \
--logging.file.max-history=2 ①
...
$ ls -ltr target/logs

total 24
-rw-r--r-- 1 jim staff 80 Mar 29 17:52 mylog.log.2020-03-29-17:52:58.0.log ①
-rw-r--r-- 1 jim staff 80 Mar 29 17:52 mylog.log.2020-03-29-17:52:59.0.log ①
-rw-r--r-- 1 jim staff 80 Mar 29 17:53 mylog.log
```

① specifying `logging.file.max-history` limited number of historical logfiles. Oldest files exceeding the criteria are deleted.

97.14. logging.file.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.file.total-size-cap`. Thus historical files with indexes 1 and 2 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.jar \
--spring.profiles.active=rollover \
--logging.file.name=target/logs/mylog.log \
--logging.file.max-size=1000 \
--logging.pattern.rolling-file-name='${logging.file.name}.%d{yyyy-MM-dd}.\%i'.log \
--logging.file.max-history=20 \
--logging.file.total-size-cap=3500 ①
...
$ ls -ltr target/logs

total 32 ②
-rw-r--r-- 1 jim staff 1040 Mar 29 18:09 mylog.log.2020-03-29.2.log ①
-rw-r--r-- 1 jim staff 1040 Mar 29 18:10 mylog.log.2020-03-29.3.log ①
-rw-r--r-- 1 jim staff 1040 Mar 29 18:10 mylog.log.2020-03-29.4.log ①
-rw-r--r-- 1 jim staff 160 Mar 29 18:10 mylog.log ①
```

① `logging.file.total-size-cap` constrains current plus historical files retained

② historical files with indexes 1 and 2 were deleted to stay below file size limit

97.15. logging.file.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.file.total-size-cap`.

Total Size Limit (without Index) Example

```
$ java -jar target/appconfig-logging-example-*-.SNAPSHOT.jar \
--spring.profiles.active=rollover \
--logging.file.name=target/logs/mylog.log \
--logging.file.max-size=100 \
--logging.pattern.rolling-file-name='${logging.file.name}.%d{yyyy-MM-dd
-HH:mm:ss}.%i'.log \
--logging.file.max-history=200 \
--logging.file.total-size-cap=500 ①
...
$ ls -ltr target/logs
James-MacBook-Pro.local: Sun Mar 29 18:33:41 2020

total 56
-rw-r--r-- 1 jim staff 79 Mar 29 18:33 mylog.log.2020-03-29-18:33:32.0.log ①
-rw-r--r-- 1 jim staff 79 Mar 29 18:33 mylog.log.2020-03-29-18:33:33.0.log ①
-rw-r--r-- 1 jim staff 79 Mar 29 18:33 mylog.log.2020-03-29-18:33:34.0.log ①
-rw-r--r-- 1 jim staff 79 Mar 29 18:33 mylog.log.2020-03-29-18:33:35.0.log ①
-rw-r--r-- 1 jim staff 80 Mar 29 18:33 mylog.log.2020-03-29-18:33:36.0.log ①
-rw-r--r-- 1 jim staff 80 Mar 29 18:33 mylog.log.2020-03-29-18:33:37.0.log ①
-rw-r--r-- 1 jim staff 80 Mar 29 18:33 mylog.log ①
```

① `logging.file.total-size-cap` constrains current plus historical files retained



The `logging.file.total-size-cap` value—if specified—must be larger than the `logging.file.max-size` constraint. Otherwise the file appender will not be activated.

Chapter 98. 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

98.1. 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

98.2. 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.

98.3. 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)

98.4. 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`

98.5. 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.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`

98.6. 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.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 99. 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 100. 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 was 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. 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 © 2021 jim stafford (jim.stafford@jhu.edu)

Chapter 101. Introduction

101.1. 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.

101.2. 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.

101.3. 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.

101.4. 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

101.5. 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 and AssertJ-based tests
6. implement unit integration tests using a Spring context
7. implement mocks (using Mockito) into a Spring context for use with unit integration tests
8. execute tests using Maven Surefire plugin

Chapter 102. 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

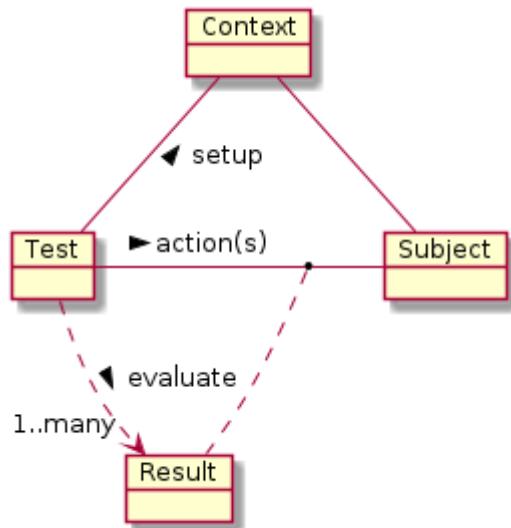


Figure 19. Basic Test Concepts

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).

102.1. 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](#). 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".

102.2. 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".

102.3. 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.

102.4. 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.

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 103. 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

103.1. 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.

```
[INFO] \- org.springframework.boot:spring-boot-starter-test:jar:2.2.1.RELEASE:test
[INFO]   +- org.springframework.boot:spring-boot-test:jar:2.2.1.RELEASE:test
[INFO]   +- org.springframework.boot:spring-boot-test-
autoconfigure:jar:2.2.1.RELEASE:test
[INFO]   +- org.springframework:spring-test:jar:5.2.1.RELEASE:test

[INFO]   +- org.junit.jupiter:junit-jupiter:jar:5.5.2:test
[INFO]     | +- org.junit.jupiter:junit-jupiter-api:jar:5.5.2:test
[INFO]     |   +- org.opentest4j:opentest4j:jar:1.2.0:test
[INFO]     |   \- org.junit.platform:junit-platform-commons:jar:1.5.2:test
[INFO]     | +- org.junit.jupiter:junit-jupiter-params:jar:5.5.2:test
[INFO]     | \- org.junit.jupiter:junit-jupiter-engine:jar:5.5.2:test

[INFO]   +- org.junit.vintage:junit-vintage-engine:jar:5.5.2:test
[INFO]     | +- org.apiguardian:apiguardian-api:jar:1.1.0:test
[INFO]     | +- org.junit.platform:junit-platform-engine:jar:1.5.2:test
[INFO]     | \- junit:junit:jar:4.12:test

[INFO]   +- org.mockito:mockito-junit-jupiter:jar:3.1.0:test
[INFO]   +- org.mockito:mockito-core:jar:3.1.0:test
[INFO]     | +- net.bytebuddy:byte-buddy:jar:1.10.2:test
[INFO]     | +- net.bytebuddy:byte-buddy-agent:jar:1.10.2:test
[INFO]     | \- org.objenesis:objenesis:jar:2.6:test

[INFO]   +- org.assertj:assertj-core:jar:3.13.2:test
[INFO]   +- org.hamcrest:hamcrest:jar:2.1:test

[INFO]   +- com.jayway.jsonpath:json-path:jar:2.4.0:test
[INFO]     | +- net.minidev:json-smart:jar:2.3:test
[INFO]     |   \- net.minidev:accessors-smart:jar:1.2:test
[INFO]     |       \- org.ow2.asm:asm:jar:5.0.4:test
[INFO]     | \- org.slf4j:slf4j-api:jar:1.7.29:compile

[INFO]   +- org.skyscreamer:jsonassert:jar:1.5.0:test
[INFO]     | \- com.vaadin.external.google:android-json:jar:0.0.20131108.vaadin1:test

[INFO]   +- jakarta.xml.bind:jakarta.xml.bind-api:jar:2.3.2:test
[INFO]     | \- jakarta.activation:jakarta.activation-api:jar:1.2.1:test
[INFO]     \- org.xmlunit:xmlunit-core:jar:2.6.3:test
```

103.2. 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 104. 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

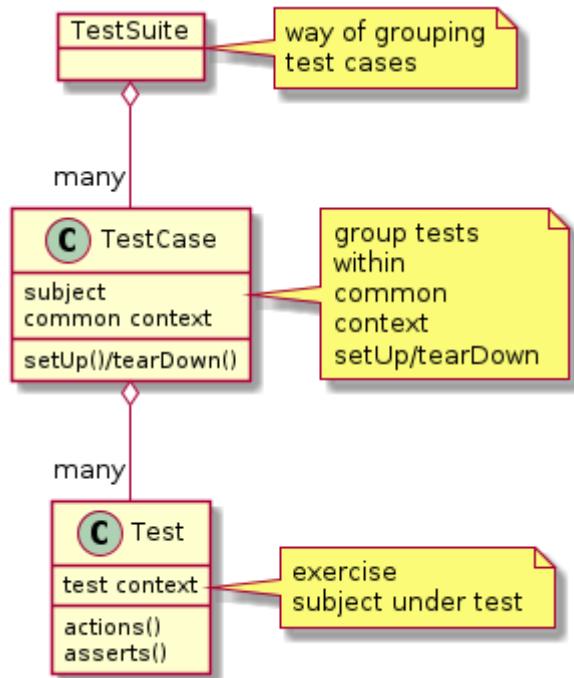


Figure 20. Basic JUnit Test Framework Constructs

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`

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

- 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  
assertEquals(4, 2+2);
```

JUnit 5/Jupiter Lambda Assertions

```
assertAll("//all get eval and reported  
        () -> assertEquals(5, 2+2),  
        () -> assertEquals(3, 2+2,  
                           ()->String.format("try%d", 2)),  
        () -> assertEquals(4, 2+2)  
    );
```

104.1. 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.

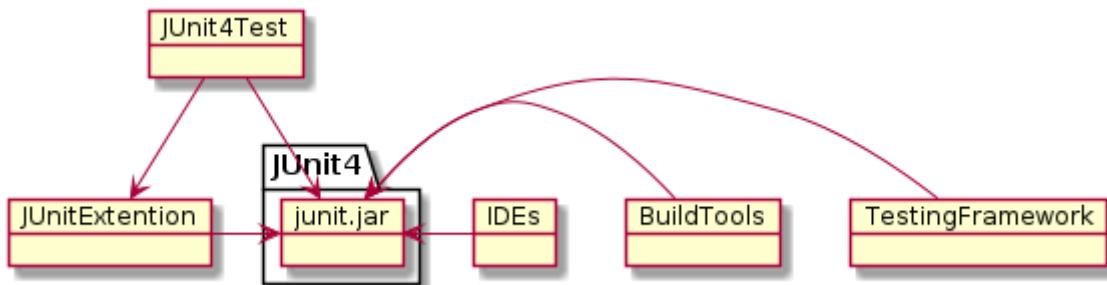


Figure 21. Everyone uses `junit.jar`

104.2. 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

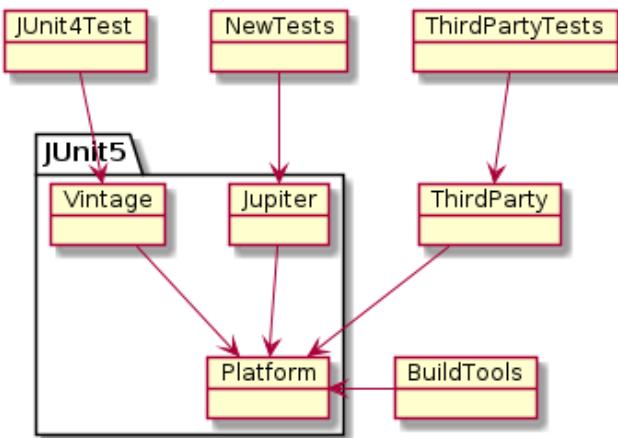


Figure 22. JUnit 5 *Modularization*



The name Jupiter was selected because it is the 5th planet from the Sun

104.3. 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.

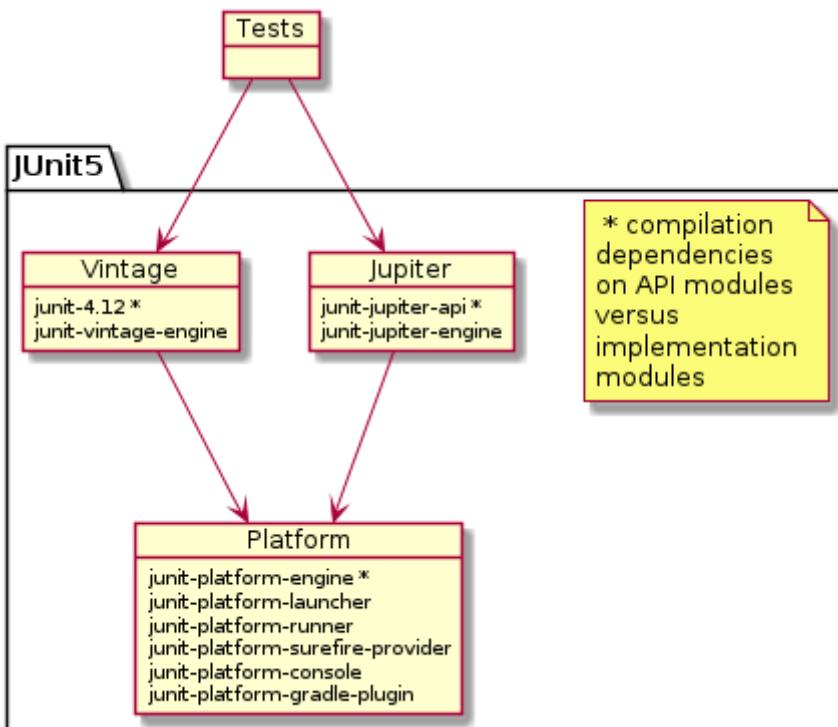


Figure 23. JUnit 5 *Module Contents*

Chapter 105. 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 106. 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.

106.1. 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

106.2. 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);
    assertTrue(String.format("problem with %d+%d", 1, 1), 1+1==2);
}
```

- @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

```
assertTrue(String.format("problem with %d+%d", 1, 1), 1+1==2);
```



Vintage requires the class and methods have public access.

106.3. 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—but 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 107. 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.

107.1. 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

107.2. 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");
    assertTrue(1+1==2, ()->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

assertTrue(1+1==2, ()->String.format("problem with %d+%d",1,1));

107.3. 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 108. 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.

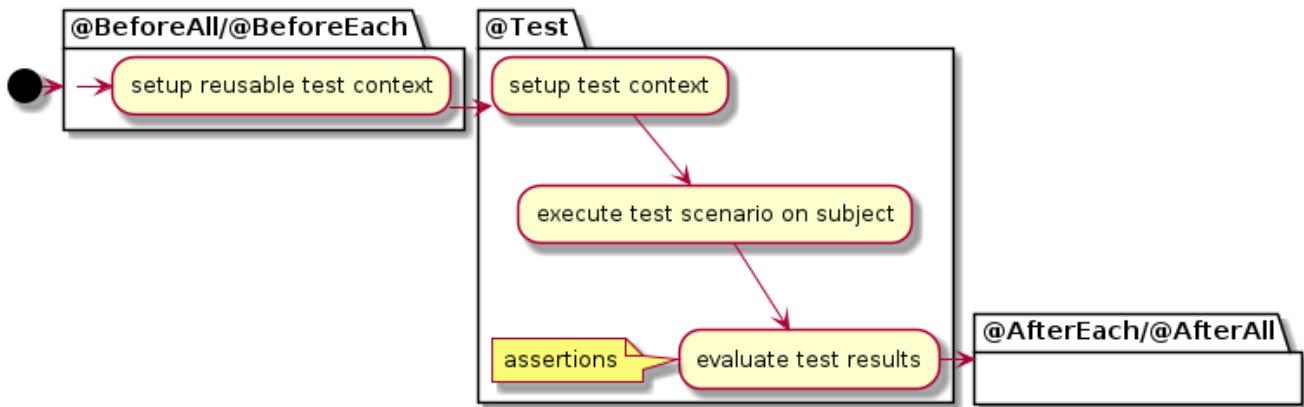


Figure 24. Assertions are Key Point in Tests

108.1. 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.



I found the following article quite helpful: [Hamcrest vs AssertJ Assertion Frameworks - Which One Should You Choose?, 2017](#) by Yuri Bushnev. Many of his data points are years out of date—but the facts he brings up with the core design of Hamcrest and AssertJ are still true and enlightening.

108.1.1. 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](#).

108.1.2. 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.*;  
...  
    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.

108.1.3. 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(beaver.getFirstName()).isEqualTo("Jerry"); ①
```

① `assertThat` is a builder of assertion factories and `isEqual` executes an assertion in chain

Custom extensions are accomplished by creating a new builder factory at the start of the 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.

108.2. 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

108.3. 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

108.3.1. 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
    //JUnit 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.

108.4. 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.

108.4.1. 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.

JUnit Jupiter Multiple Assertion Support

```
@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

108.4.2. 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

108.5. 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.

108.5.1. 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 this point forward JUnit assertions do not provide any direct support to inspect the Exception.

```
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

108.5.2. AssertJ Exception Handling Support

AssertJ has an Exception testing capability that is similar to JUnit Vintage — 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.

```
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

```
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

108.6. Asserting Dates

AssertJ has built-in support for date assertions. We have to add a separate library to gain date matchers for Hamcrest.

108.6.1. AssertJ Date Handling Support

The following shows an example of AssertJ's built-in, natural-language support for Dates.

```
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

108.6.2. 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 109. 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.

109.1. 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

109.2. 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)

109.3. 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.

109.4. Mockito Example Test

Basic Mockito 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 110. 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

110.1. 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

110.2. 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
            //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

110.3. 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 25. Example BDD Syntax Output

Chapter 111. 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.

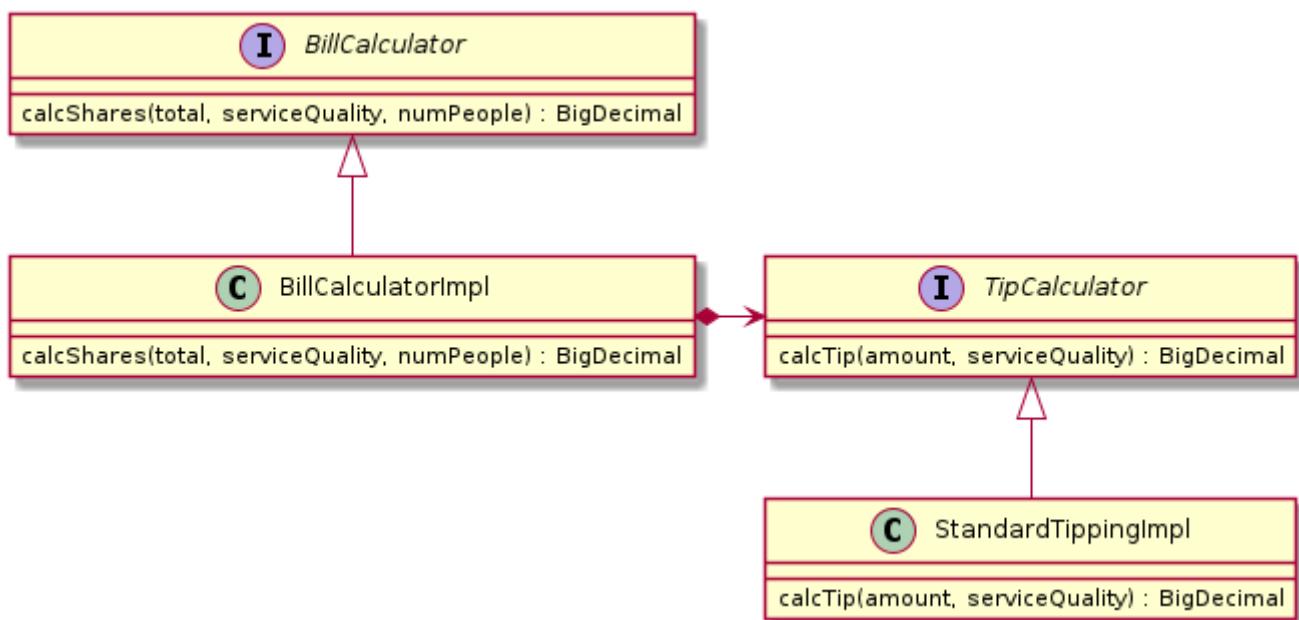


Figure 26. Tipping Example Class Model

Chapter 112. 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.

112.1. Review: POJO Unit Test Setup

Review: POJO Unit Test Setup

```
@DisplayNameGeneration(DisplayNameGenerator.ReplaceUnderscores.class) ①
@Repository("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

112.2. 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

112.3. 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

112.4. 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

112.5. Alternative Mocked Unit Test

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 and injecting Mocks
     */
    @InjectMocks ①
    BillCalculatorImpl billCalculator;
```

① instantiates and injects out subject under test

Chapter 113. 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.

113.1. 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 preparing these lecture notes. The [Spring Boot Testing Features](#) 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.

113.2. `@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

113.3. 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);  
    }  
}
```

113.4. 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)

113.5. 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 {
```

113.6. 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;
```

113.7. 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

113.8. 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);
    }
}
```

113.9. Example @SpringBootTest Unit Integration Test 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 ::          (v2.4.2)

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
```

113.10. 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 114. 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

114.1. 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 115. 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
```

115.1. 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.

115.2. 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

115.3. 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.

```
$ 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

115.4. 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

115.5. 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 116. 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

Race Registration/Results Assignment 1

copyright © 2021 jim stafford (jim.stafford@jhu.edu)

- Changes
 - 2021-09-21: debug profile changed to trace logging in grading criteria

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. 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 are overlapped. 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-starters/race-starters](#). 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
- 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-race-beanfactory
|   |-- pom.xml
|   |-- race-beanfactory-app
|   |-- race-beanfactory-iface
|   `-- race-beanfactory-swim
|-- assignment1-race-propertiesource
|   |-- pom.xml
|   `-- src
|-- assignment1-race-configprops
|   |-- pom.xml
|   `-- src
|-- assignment1-race-autoconfig
|   |-- pom.xml
|   |-- race-autoconfig-app
|   |-- race-autoconfig-iface
|   |-- race-autoconfig-run
|   |-- race-autoconfig-starter
|   `-- race-autoconfig-swim
|-- assignment1-race-logging
|   |-- pom.xml
|   `-- src
|-- assignment1-race-testing
|   |-- pom.xml
|   `-- src
`-- pom.xml
```

Chapter 117. Assignment 1a: App Config

- Changes

- 2021-09-10 - corrected name of application-production.properties file in Property Source Configuration

117.1. @Bean Factory Configuration

117.1.1. 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

117.1.2. 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.

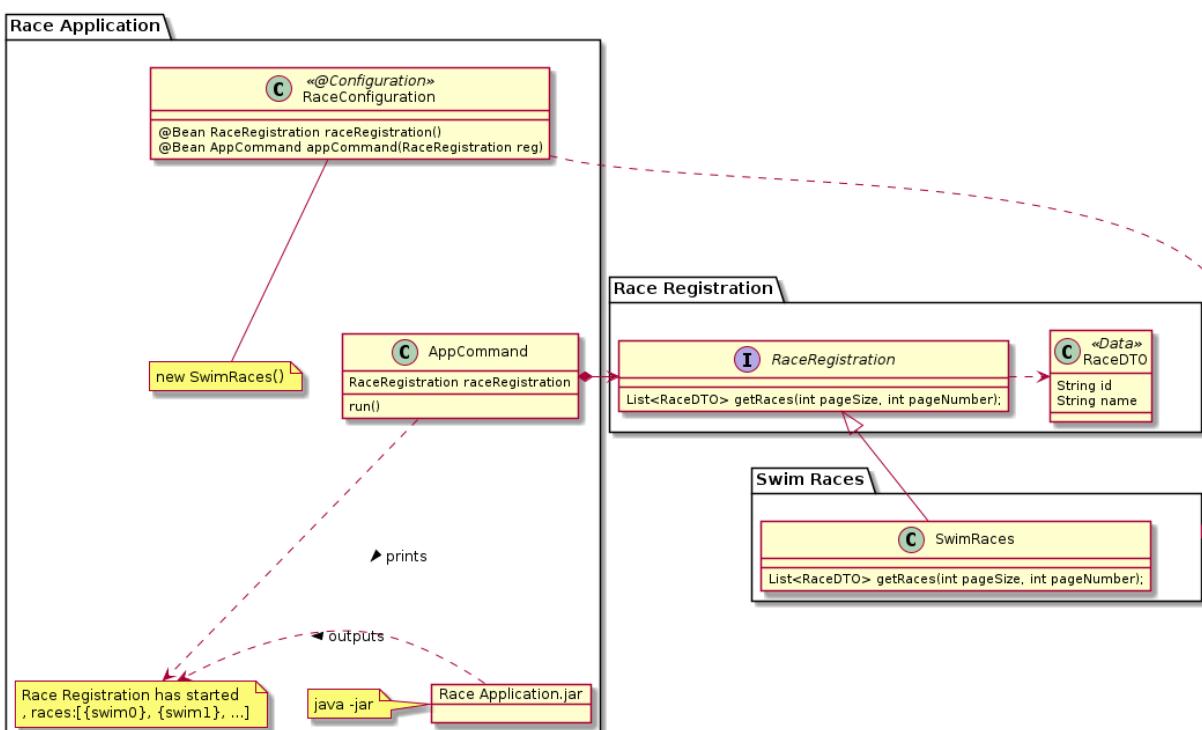


Figure 27. @Bean Factory Configuration

117.1.3. Requirements

1. Create an interface module with
 - a. a `RaceDTO` class with `id` and `name` properties. This is a simple data class.
 - b. a `RaceRegistration` interface with a `getRaces()` method. This method should accept `pageSize` and `pageNumber` and return a list of `RaceDTO` instances.
2. Create a Swim implementation module with
 - a. a Swim implementation of the `RaceRegistration` interface that returns a list of races that equal `pageSize` and have a name with "swim" within it (e.g., "swim0", "swim1", etc.). The `RaceDTO` instances should have a name with "swim" within it (e.g., "swim0", "swim1", etc.).
3. Create an application module with
 - a. a class (e.g., `CommandLineRunner`) that
 - i. has the `RaceRegistration` component injected using **constructor injection**
 - ii. at startup
 - A. calls the `RaceRegistration` for a page of races
 - B. prints a startup message with a page of race names
 - 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 `RaceRegistration` swim implementation
 - ii. one `@Bean` factory method to instantiate the `AppCommand` injected with a `RaceRegistration` bean (not a POJO)
 - c. a `@SpringBootApplication` class that uses 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.

117.1.4. 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 RaceRegistration component
 - b. whether the @Configuration class successfully instantiates the startup message component injected with a RaceRegistration component.

117.1.5. 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-starters/race-starters/assignment1-race-beanfactory](#). Modify Maven groupId and Java package if used.

117.2. Property Source Configuration

117.2.1. 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

117.2.2. 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.

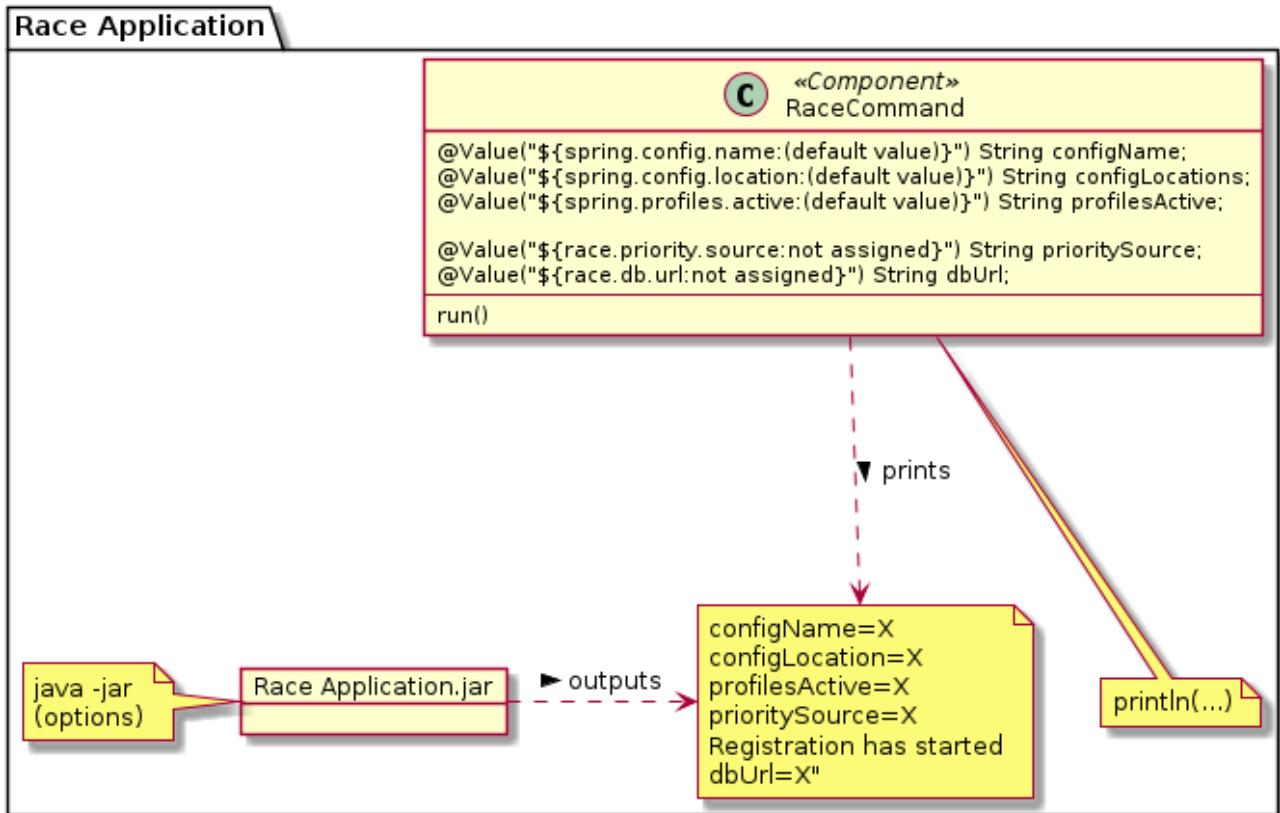


Figure 28. Property Source Configuration

This assignment involves very little - to 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.

117.2.3. Requirements

- Given the following unmodified `@Component` property `@Value` injections, unmodified `application.properties` file, and fixed property file sources (available in complete source form in the "assignment starter") ...
 - `@Component` Property `@Value` Injections (*this cannot be modified*)

```
public class RaceCommand implements CommandLineRunner {  
    @Value("${spring.config.name:(default value)}") String configName; ①  
    @Value("${spring.config.location:(default value)}") String configLocations;  
    @Value("${spring.profiles.active:(default value)}") String profilesActive;  
  
    @Value("${race.priority.source:not assigned}") String prioritySource;  
    @Value("${race.db.url:not assigned}") String dbUrl;
```

① place these exact @Value property injections into your @Component

- b. application.properties file (*this cannot be modified*)

```

#application.properties
race.priority.source=application.properties
race.db.user=user
race.db.port=00000
①
race.db.url=mongodb://${race.db.user}:${race.db.password}@${race.db.host}:${race
.db.port}/test?authSource=admin

```

① `race.db.url` is built from several property placeholders. `password` is not specified.

c. Property File Sources (*this will be set on the command line and cannot be modified*)

```

classpath:/ ①
application.properties
application-default.properties
application-dev.yml ③
application-production.properties
file:src/test/resources/ ②
application-dev1.properties
application-site1.properties

```

① classpath-based files are in the source tree, but referenced in the JAR/classpath at runtime

② file-based references are simulating location-specific resources not within the application

③ this file must be a YAML file



application-dev.yml must be a YAML file

application-dev.yml must be expressed using YAML syntax

d. Base Command (*the parameters to the program cannot be modified*)

```

$ java -jar target/assignment1-race-propertiesource-1.0-SNAPSHOT.jar
--spring.config.location=classpath:/,file:src/test/resources/ ①

```

① this is the base command for 3 specific commands that specify profiles active

```
--spring.profiles.active= ①
```

① the following 3 commands will supply a different value for this property

e. Populate the property and YAML files so that the scenarios in the following paragraph are satisfied.

Supply/fill-in the Configuration Files



The meat of this assignment is to provide/fill in the property and YAML files so that the program prints matching information to what is below based on the changing input configuration settings.

2. When activating the following profiles, the application must output the following results

- a. No Active Profile Command Result

```
configName=(default value)
configLocation=classpath:/,file:src/test/resources/
profilesActive=(default value)
prioritySource=application-default.properties
Race Registration has started
dbUrl=mongodb://defaultUser:defaultPass@defaultHost:27027/test?authSource=admin
```

The default starter with the "base command" and "no active profile" set, produces the following by default.



```
$ java -jar target/assignment1-race-propertysource-1.0-
SNAPSHOT.jar
--spring.config.location=classpath:/,file:src/test/resources/

configName=(default value)
configLocation=classpath:/,file:src/test/resources/
profilesActive=(default value)
prioritySource=application-default.properties
Race Registration has started
dbUrl=mongodb://user:NOT_SUPPLIED@NOT_SUPPLIED:00000/test?authSo
urce=admin
```

You must supply a populated set of configuration files so that, under this option, **user:NOT_SUPPLIED@NOT_SUPPLIED:00000** becomes **defaultUser:defaultPass@defaultHost:27027**. You are not allowed to change the Java attribute declarations nor change the **application.properties** file.

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

- b. dev,dev1 Active Profile Command Result

```
--spring.profiles.active=dev,dev1
```

```
configName=(default value)
configLocation=classpath:/,file:src/test/resources/
profilesActive=dev,dev1
prioritySource=dev1.properties
Race Registration has started
dbUrl=mongodb://devUser:dev1pass@127.0.0.1:17027/test?authSource=admin
```



You must supply a populated set of configuration files so that, under this option, `user:NOT_SUPPLIED@NOT_SUPPLIED:00000` becomes `devUser:dev1pass@127.0.0.1:17027`

Developer/site-specific values (e.g., dev1pass) must be provided by a property file outside of the JAR.

c. production,site1 Active Profile Command Result

```
--spring.profiles.active=production,site1
```

```
configName=(default value)
configLocation=classpath:/,file:src/test/resources/
profilesActive=production,site1
prioritySource=site1.properties
Race Registration has started
dbUrl=mongodb://productionUser:site1pass@db.site1.net:27017/test?authSource=adm
n
```



You must supply a populated set of configuration files so that, under this option, `user:NOT_SUPPLIED@NOT_SUPPLIED:00000` becomes `productionUser:site1pass@db.site1.net:27017`

Developer/site-specific values (e.g., site1pass) must be provided by a property file outside of the JAR.

3. Turn in a source tree with a complete Maven module that will build and demonstrate the `@Value` injections for the 3 different active profile settings.

117.2.4. 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 `@Value` injections were declared with default values
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` and `site1` 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

117.2.5. 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-starters/race-starters/assignment1-race-propertysource`. Modify Maven groupId and Java package if used.
3. 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=race
configLocation=(default value)
profilesActive=(default value)
prioritySource=not assigned
Race Registration has started
dbUrl=not assigned
```

117.3. Configuration Properties

117.3.1. Purpose

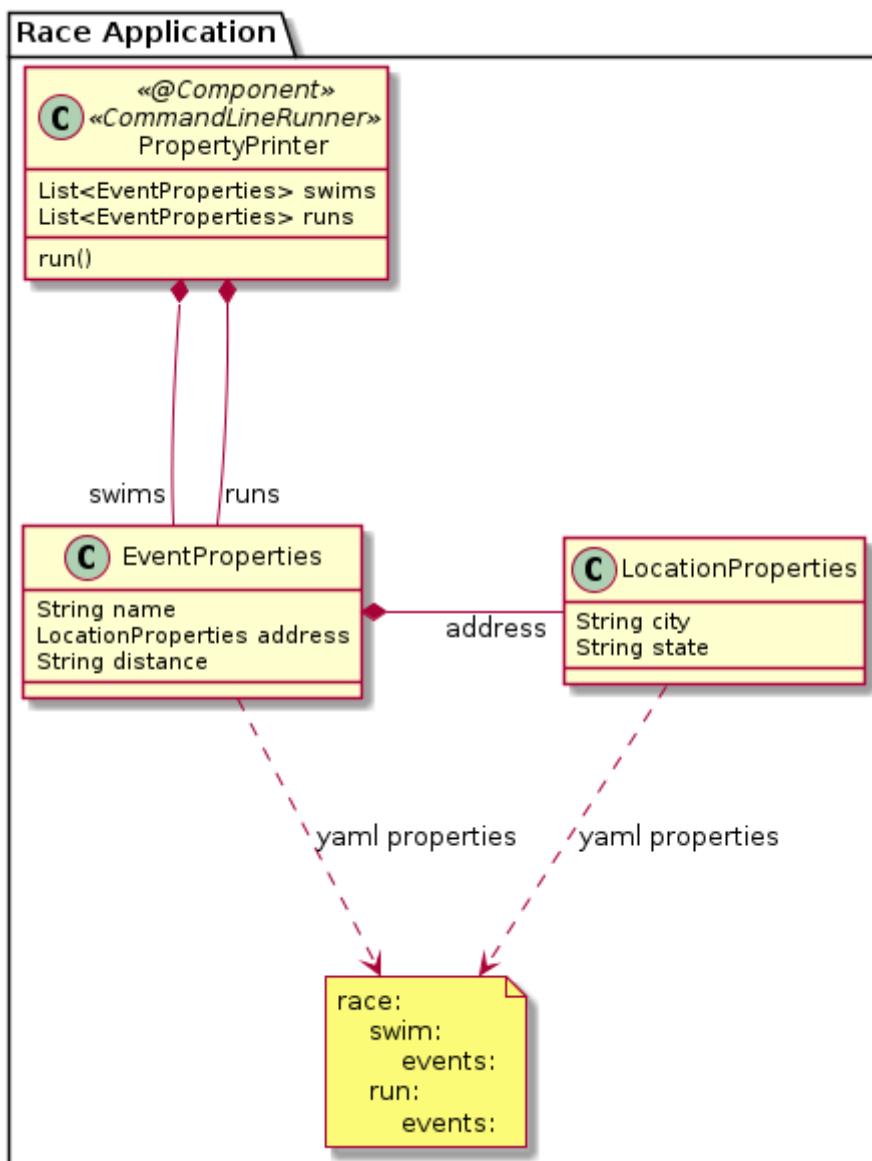
In this portion of the assignment, you will demonstrate your knowledge of injecting properties into a `@ConfigurationProperties` class to be injected into components - to encapsulate the runtime configuration of the component(s). You will:

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

117.3.2. 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.



117.3.3. Requirements

1. Given the following read-only property classes, application.yml file, and `@Component` ...
 - a. read-only property classes

```
@Value  
public class EventProperties {  
    private String name;  
    private LocationProperties address;  
    private String distance;  
}
```

```
@Value  
public class LocationProperties {  
    //@NotBlank  
    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.

b. `application.yml` YAML file

```
race:  
  swim:  
    events:  
      - name: OCNJ Masters Swim  
        address:  
          city: Ocean City  
          state: NJ  
        distance: 1 mi  
  run:  
    events:  
      - name: Baltimore 5k  
        address:  
          city: Baltimore  
          state: MD  
        distance: 5K  
      - name: Maryland CE Marathon  
        address:  
          city: Severna Park  
          state: MD  
        distance: 26.2 mi  
      - name: Boston Marathon  
        distance: 26.2 mi  
        address:  
          state: MA
```

c. `@Component` with constructor injection

```

//@Component
@RequiredArgsConstructor
public class PropertyPrinter implements CommandLineRunner {
    private final List<EventProperties> swims;
    private final List<EventProperties> runs;

    @Override
    public void run(String... args) throws Exception {
        System.out.println("swims:" + format(swims));
        System.out.println("runs:" + format(runs));
    }

    private String format(List<EventProperties> events) {
        return String.format("%s", events.stream()
            .map(r->"*" + r.toString())
            .collect(Collectors.joining(System.lineSeparator(), System.
lineSeparator(), "")));
    }
}

```

2. When running the application, a `@ConfigurationProperties` bean will be created to represent the contents of the YAML file as two separate `List<EventProperties>` objects, injected into the `@Component`, and will output the following.

```

swims:
*EventProperties(name=OCNJ Masters Swim, address=LocationProperties(city=Ocean
City, state=NJ), distance=1 mi)
runs:
*EventProperties(name=Baltimore 5k, address=LocationProperties(city=Baltimore,
state=MD), distance=5K)
*EventProperties(name=Maryland CE Marathon, address=LocationProperties(city=Severna
Park, state=MD), distance=26.2 mi)
*EventProperties(name=Boston Marathon, address=LocationProperties(city=null,
state=MA), distance=26.2 mi)

```



The "assignment starter" supplies most of the Java code needed for the `PropertyPrinter`. The focus of this assignment is to complete making that a component with two injected `@ConfigurationProperties` for `swims` and `runs`.

3. Turn in a source tree with a complete Maven module that will build and demonstrate the configuration property processing and output of this application.

117.3.4. 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 (`swim` and `run`) 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.

117.3.5. 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-starters/race-starters/assignment1-race-configprops`. Modify Maven groupId and Java package if used.
3. Ungraded Question to Ponder: What single line change (excluding import(s)) could be made to the application to validate the properties and report the following error for run events?

Alternate Output

```
*****
APPLICATION FAILED TO START
*****
```

Description:

Binding to target `org.springframework.boot.context.properties.bind.BindException`:
 Failed to bind properties under '`race.run.events`' to
`java.util.List<...EventProperties>` failed:

Property: `race.run.events[2].address.city`
 Value: `null`
 Reason: must not be blank

117.4. Auto-Configuration

117.4.1. Purpose

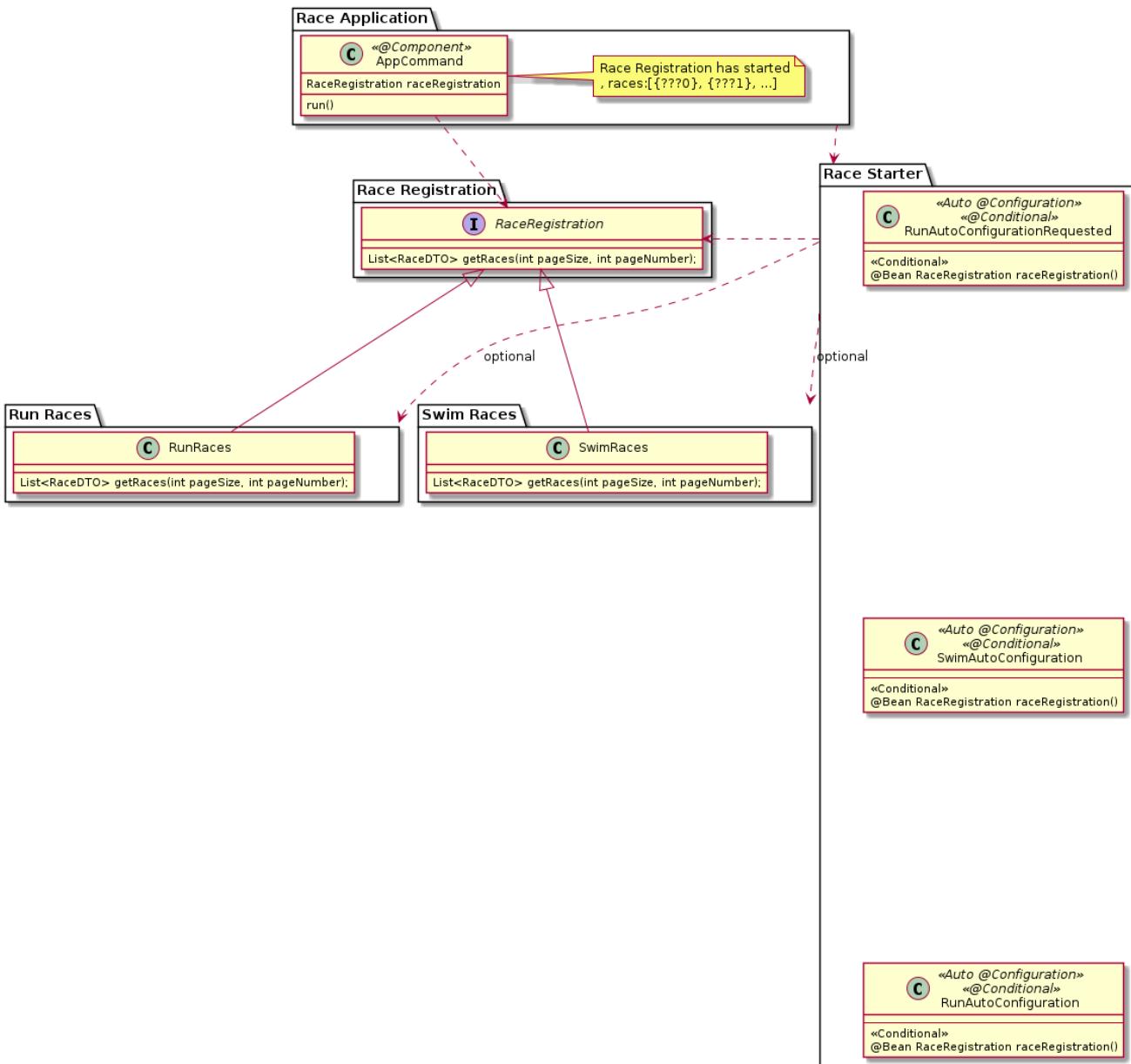
In this portion of the assignment, you will demonstrate your knowledge of developing and using Auto-Configuration classes to control the configuration of an application.

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 a `spring.factories` 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

117.4.2. 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 `RaceRegistration` out of a choice of two (2) based on the environment at runtime.



117.4.3. Requirements



Much of this can be based on your Bean Factory solution. Make a new copy of that work to implement the requirements of this assignment.

1. Create a "Race Registration Interface" module
 - a. Add an interface to return several races as **RaceDTO** instances
2. Create a "Swim Registration" implementation module
 - a. Add an implementation of the interface to return "swim" races
3. Create a "Run Registration" implementation module
 - a. Add an implementation of the interface to return "run" races
4. Create an Application Module with a **@SpringBootApplication** class
 - a. Add a **@Component** class that gets injected with a **RaceRegistration** bean and prints "RaceRegistration has started" with the names of races coming from the injected bean.

- b. Remove all Maven dependencies on the "Race Registration" interface and implementation modules.



At this point you are have mostly repeated the bean factory solution except that you have eliminated the `@Bean` factory for the `RaceRegistration` in the Application module, added a Run implementation option, and removed a few Maven module dependencies.

5. Create a "Race Registration Starter" Module

- a. Add a dependency on the "Race Registration" interface module
- b. Add a dependency on the race implementation modules and make them "**optional**" (this is **important**) so that the application module will need to make an explicit dependency on the implements for them to be path of the runtime classpath.
- c. Add conditional Auto-`@Configuration` classes
 - i. one that provides a `@Bean` factory for the "Run Races" implementation class
 - A. Make this conditional on the presence of the "Run Registration" class(es) being available on the classpath
 - ii. one that provides a `@Bean` factory for the "Swim Races" implementation class
 - A. Make this conditional on the presence of the "Swim Registration" class(es) being available on the classpath
 - iii. A third that provides another `@Bean` factory for the "Run Races" implementation class
 - A. Make this conditional on the presence of the "Run Registration" class(es) being available on the classpath
 - B. Make this also conditional on the property `races.registration.type` having the value of `run`.
- d. Set the following priorities for the Auto-`@Configuration` classes
 - i. make the "Run Races"/property Auto-`@Configuration` the highest priority
 - ii. make the "Swim Races" Auto-`@Configuration` factory the next highest priority
 - iii. make the "Run Races" Auto-`@Configuration` 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 "Swim Races" and "Run Races" `@Bean` factories if the property `races.registration.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 `@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 `@Configuration` classes and not nested classes `@Configuration` classes within them.

6. Modify the Application module pom

- a. Add a dependency on the Starter Module
- b. Create a profile (`swim`) that adds a direct dependency on the "Swim Registration" module. The "assignment starter" provides an example of this.
- c. Create a profile (`run`) that adds a direct dependency on the "Run Registration" 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 clean package
$ mvn dependency:list -f race-autoconfig-app/ | egrep 'race-autoconfig.*jar'
info.ejava-student.assignment1.race:race-autoconfig-starter:jar:1.0-
SNAPSHOT:compile
info.ejava-student.assignment1.race:race-autoconfig-iface:jar:1.0-
SNAPSHOT:compile
①
$ java -jar race-autoconfig-app/target/race-autoconfig-app*.jar

*****
APPLICATION FAILED TO START
*****
Description:
Parameter 0 of method ... required a bean of type '...RaceRegistration' that
could not be found.
Action:
Consider defining a bean of type '...RaceRegistration' in your configuration.
```

- ① no registration implementation jars in dependency classpath

- b. `swim` only Maven profile active and no properties provided

```

$ mvn clean package -P swim
$ mvn dependency:list -P swim -f race-autoconfig-app/ | egrep 'race-
autoconfig.*jar'
info.ejava-student.assignment1.race:race-autoconfig-starter:jar:1.0-
SNAPSHOT:compile
info.ejava-student.assignment1.race:race-autoconfig-swim:jar:1.0-
SNAPSHOT:compile ①
info.ejava-student.assignment1.race:race-autoconfig-iface:jar:1.0-
SNAPSHOT:compile

$ java -jar race-autoconfig-app/target/race-autoconfig-app*.jar
Race Registration has started, races:[{swim0}, {swim1}, {swim2}]

```

① `swim` registration JAR in dependency classpath

- c. `run` only Maven profile active and no properties provided

```

$ mvn clean package -P run
$ mvn dependency:list -P run -f race-autoconfig-app/ | egrep 'race-
autoconfig.*jar'
info.ejava-student.assignment1.race:race-autoconfig-starter:jar:1.0-
SNAPSHOT:compile
info.ejava-student.assignment1.race:race-autoconfig-iface:jar:1.0-
SNAPSHOT:compile
info.ejava-student.assignment1.race:race-autoconfig-run:jar:1.0-SNAPSHOT:compile
①

$ java -jar race-autoconfig-app/target/race-autoconfig-app*.jar
Race Registration has started, races:[{run0}, {run1}, {run2}]

```

① `run` registration JAR in dependency classpath

- d. `swim` and `run` Maven profiles active

```

$ mvn clean install -P run,swim
$ mvn dependency:list -P run,swim -f race-autoconfig-app/ | egrep 'race-
autoconfig.*jar'
info.ejava-student.assignment1.race:race-autoconfig-starter:jar:1.0-
SNAPSHOT:compile
info.ejava-student.assignment1.race:race-autoconfig-iface:jar:1.0-
SNAPSHOT:compile
info.ejava-student.assignment1.race:race-autoconfig-swim:jar:1.0-
SNAPSHOT:compile ①
info.ejava-student.assignment1.race:race-autoconfig-run:jar:1.0-SNAPSHOT:compile
②

$ java -jar race-autoconfig-app/target/race-autoconfig-app*.jar
Race Registration has started, races:[{swim0}, {swim1}, {swim2}]

```

① `swim` registration JAR in dependency classpath

② `run` registration JAR in dependency classpath

- e. `swim` and `run` Maven profiles active and Spring property `races.registration.type=run`

```
$ mvn clean install -P run,swim ①
$ java -jar race-autoconfig-app/target/race-autoconfig-app*.jar
--races.registration.type=run ②
Race Registration has started, races:[{run0}, {run1}, {run2}]
```

① `swim` and `run` registration JARs in dependency classpath

② `races.registration.type` property supplied with `true` value

- f. `swim` and `run` Maven profiles active and Spring property `races.registration.active=false`

```
$ mvn clean install -P run,swim ①
$ java -jar race-autoconfig-app/target/race-autoconfig-app*.jar
--races.registration.active=false ②
*****
APPLICATION FAILED TO START
*****
Description:
Parameter 0 of method appCommand in ...RaceConfiguration required a bean of type
'...RaceRegistration' that could not be found.
Action:
Consider defining a bean of type '...RaceRegistration' in your configuration.
```

① `swim` and `run` registration JARs in dependency classpath

② `races.registration.type` property supplied with `false` 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.

117.4.4. 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 `spring.factories` 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

117.4.5. Additional Details

1. A starter project is available in [assignment-starters/race-starters/assignment1-race-autoconfig](#). Modify Maven groupId and Java package if used.

Chapter 118. Assignment 1b: Logging

118.1. Application Logging

118.1.1. 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

118.1.2. 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.

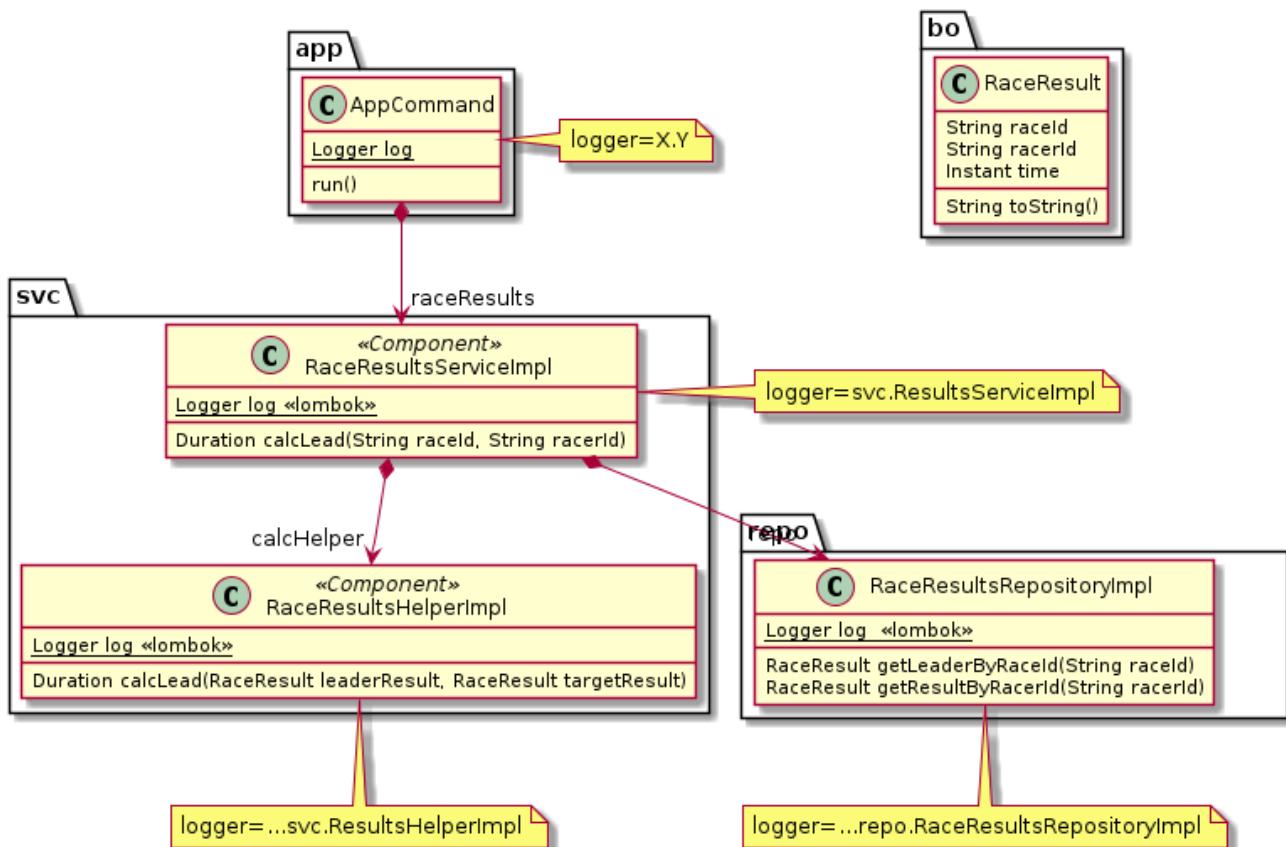


Figure 29. Application Logging

118.1.3. 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 race result.

1. Create several components in different Java sub-packages
 - a. an `AppCommand @Component` in the `app` Java sub-package
 - b. a `RaceResultsServiceImpl @Component` in the `svc` Java sub-package
 - c. a `RaceResultsHelperImpl @Component` in the `svc` Java sub-package
 - d. a `RaceResultsRepositoryImpl @Component` in the `repo` Java sub-package
2. Implement a chain of calls from the `AppCommand @Component run()` method through the other components.

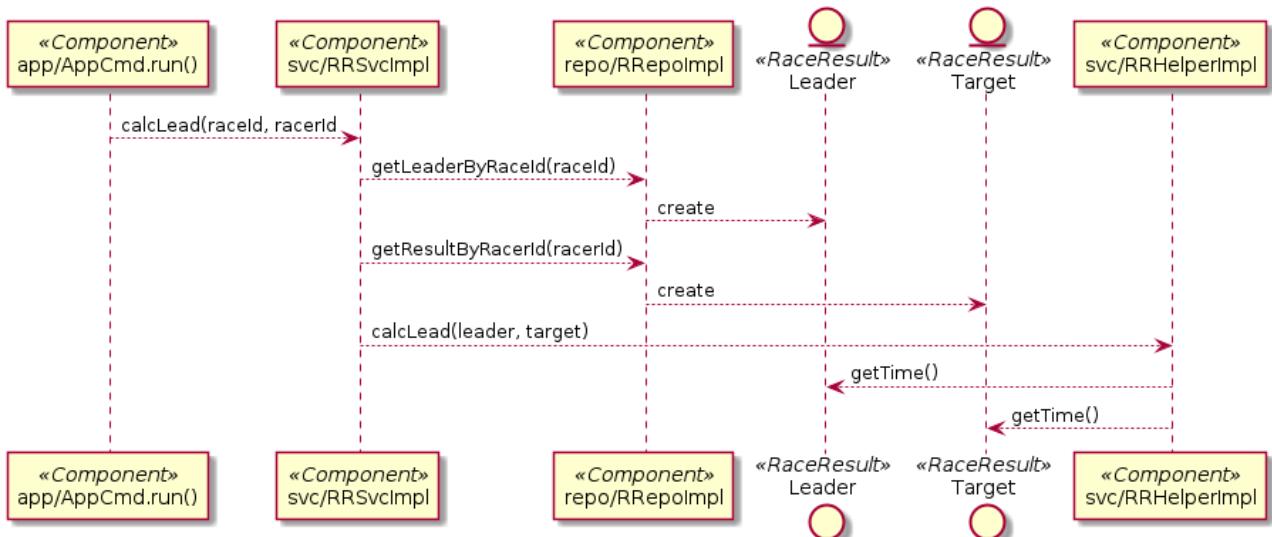


Figure 30. Required Call Sequence

- a. `AppCommand.run()` calls `RaceResultsServiceImpl.calcLead(raceId, racerId)` with a `raceId` and `racerId` to determine how far the racer is behind the race leader .
- b. `RaceResultsServiceImpl.calcLead(raceId, racerId)` calls `RaceResultsRepositoryImpl(getLeaderByRaceId(raceId))` and `getResultByRacerId(racerId))` to get `RaceResults`
 - i. `RaceResultsRepositoryImpl` can create transient instances with provided Ids and random remaining properties
- c. `RaceResultsServiceImpl.calcLead(raceId, racerId)` also calls `ResultsHelper.calcLead()` to get the time delta between the two `RaceResults`
- d. `RaceResultsHelperImpl.calcLead(leader, target)` calls `RaceResult.getTime()` on the two provided `RaceResult` instances to determine the time duration delta
3. Implement a `toString()` method in `RaceResult` that includes the `raceId`, `racerId`, and `time` 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 the 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 `RaceResultsRepositoryImpl` logging events are all TRACE
 - b. the severity of `RaceResultsHelperImpl.calcLead()` logging events are DEBUG and TRACE (there must be at least two — one of each and no other levels)
 - c. the severity of `RaceResultsServiceImpl.calcLead()` 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 race results information in log statements
- a. Leverage the LSF4J parameter formatting syntax when implementing the log
 - b. For each of the INFO statements, include only the `RaceResult` property values (i.e., `raceId`, `racerId`, `timeDelta`)
 - c. For each of the TRACE statements, include the output of `RaceResult.toString()` method.
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., 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

118.1.4. 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` 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

118.1.5. 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 [assignment-starters/race-starters/assignment1-race-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`
 - `appenders` and `trace`

Modify Maven groupId and Java package if used.

118.2. Logging Efficiency

118.2.1. Purpose

In this portion of the assignment, you will demonstrate your knowledge of making suppressed logging efficient. You will:

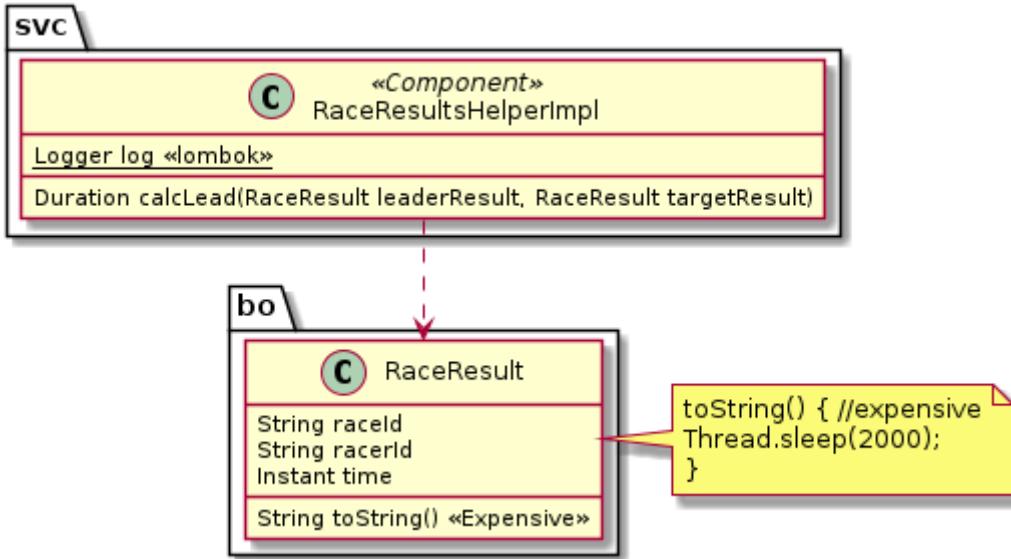
1. efficiently bypass log statements that do not meet criteria

118.2.2. Overview

In this portion of the assignment, you are going to increase the cost of calling `toString()` on the business 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.



118.2.3. Requirements

1. Update the `toString()` method in `RaceResult` to be expensive to call
 - a. artificially insert a 2 second 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

118.2.4. 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 2 seconds only for TRACE verbosity 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

118.2.5. 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.

118.3. Appenders and Custom Log Patterns

118.3.1. 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

118.3.2. 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.

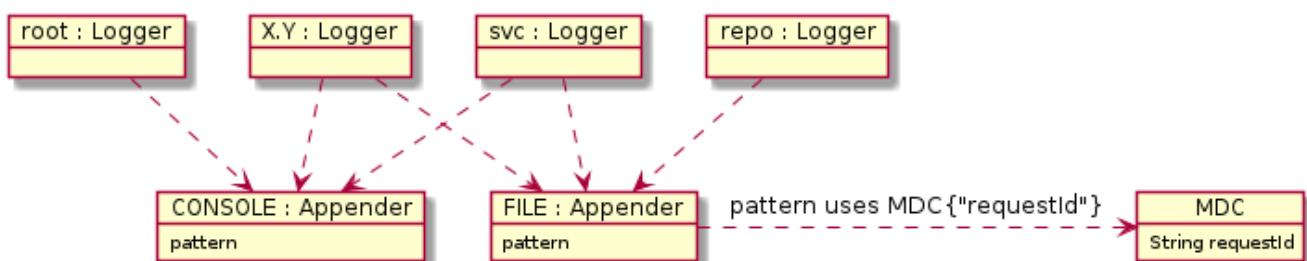


Figure 31. Appenders and Custom Log Patterns



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.

118.3.3. Requirements

1. Declare two Appenders as part of a custom Logback configuration
 - a. CONSOLE to output to stdout
 - b. FILE 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

3. Add an `appenders` profile that

- a. automatically enacts the requirements above
 - b. activates logging events at DEBUG severity and up for all loggers with your application
4. Add a `requestId` property to the Mapped Diagnostic Context (MDC)

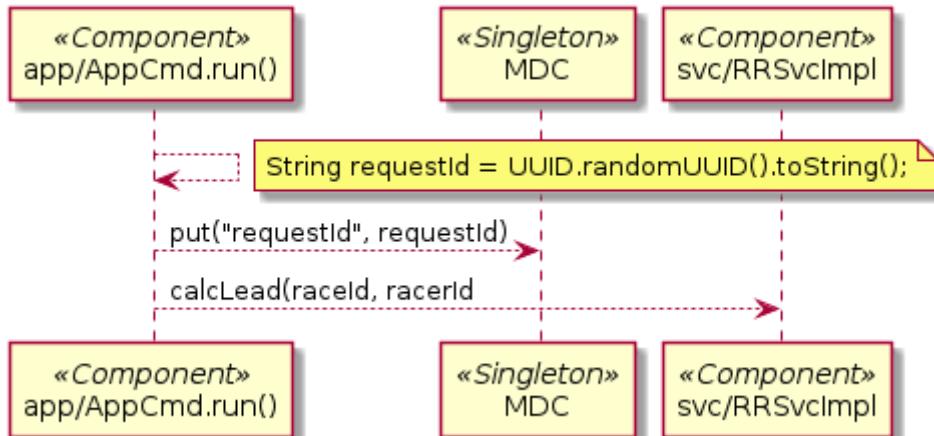


Figure 32. Initialize Mapped Diagnostic Context (MDC)

- a. generate a random/changing value
 - b. insert the value prior to the first logging statement — in the `AppCommand @Component`.
5. Declare a custom logging pattern in the `appenders` profile that includes the MDC `requestId` value in each log statements written by the FILE Appender
- a. The MDC `requestId` is only output by the FILE Appender
 - b. The MDC `requestId` is **not** output by the CONSOLE Appender
6. Add an additional `trace` profile that
- a. activates logging events at TRACE severity and up for all loggers with your application
 - b. adds method and line number information to all entries in the FILE Appender but not the CONSOLE Appender



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.

7. Apply the `appenders` profile to
- a. output logging events at DEBUG 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

118.3.4. 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 where put in place when activating the `appenders` profile
 - b. whether your required logging configurations where put in place when activating the `appenders` and `trace` profiles

118.3.5. 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"/>
    ...

```

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

```
12:05:32.232 INFO -- [c430adfb-7c61-4b99-8b83-a722e867a14f]
RaceResultsServiceImpl : calculating lead for raceId=2021-01, racerId=0115 ①
12:05:32.237 DEBUG -- [c430adfb-7c61-4b99-8b83-a722e867a14f] RaceResultHelperImpl
: calculating delta between 744 and 0115 ②
```

① no method and line number info supplied

② **requestId** is supplied in all FILE output when **appenders** profile active

Example target/logs/appenders.log - "appenders,trace" profiles active

①

```
12:05:33.557 INFO -- [7b0aca4a-51eb-498a-8cda-3671c05fc0e6]
RaceResultsServiceImpl.calcLead:18 : calculating lead for raceId=2021-01, racerId=0115
②
12:05:33.561 TRACE -- [7b0aca4a-51eb-498a-8cda-3671c05fc0e6]
RaceResultsRepositoryImpl.getLeaderByRaceId:22 : raceId 2021-01 leader is
RaceResult{raceId='2021-01', racerId='286', time=2021-07-23T10:36:31.501801Z}
12:05:35.576 TRACE -- [7b0aca4a-51eb-498a-8cda-3671c05fc0e6]
RaceResultsRepositoryImpl.getResultByRacerId:33 : racerId 0115 is
RaceResult{raceId='2021-83', racerId='0115', time=2021-07-23T20:19:27.501810Z}
12:05:37.579 DEBUG -- [7b0aca4a-51eb-498a-8cda-3671c05fc0e6]
RaceResultHelperImpl.calcLead:12 : calculating delta between 286 and 0115
```

① method and line number info supplied

② **requestId** is supplied in all FILE output when **appenders** profile active

Chapter 119. 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 blazing fast, but lack the target runtime container environment
- tests that use Mocks keep the focus on the subject being tested, 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.

119.1. Demo

The `assignment1-race-testing` assignment starter contains a `@SpringBootApplication` main class and a some demonstration code that will execute at startup when using the `demo` profile.

RaceRegistration Demonstration

```
$ mvn package -Pdemo
15:09:32.739  INFO -- RacerRegistrationServiceImpl : racer registered:
RacerDTO(id=null, firstName=michael, lastName=phelps, dob=1985-06-30)
15:09:32.753  INFO -- RacerRegistrationServiceImpl : invalid racer: RacerDTO(id=null,
firstName=future, lastName=phelps, dob=2021-09-17), [racer.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.

119.2. Unit Testing

119.2.1. 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

119.2.2. Overview

In this portion of the assignment, you are going to implement a test case with 2 unit tests for a completed Java class.

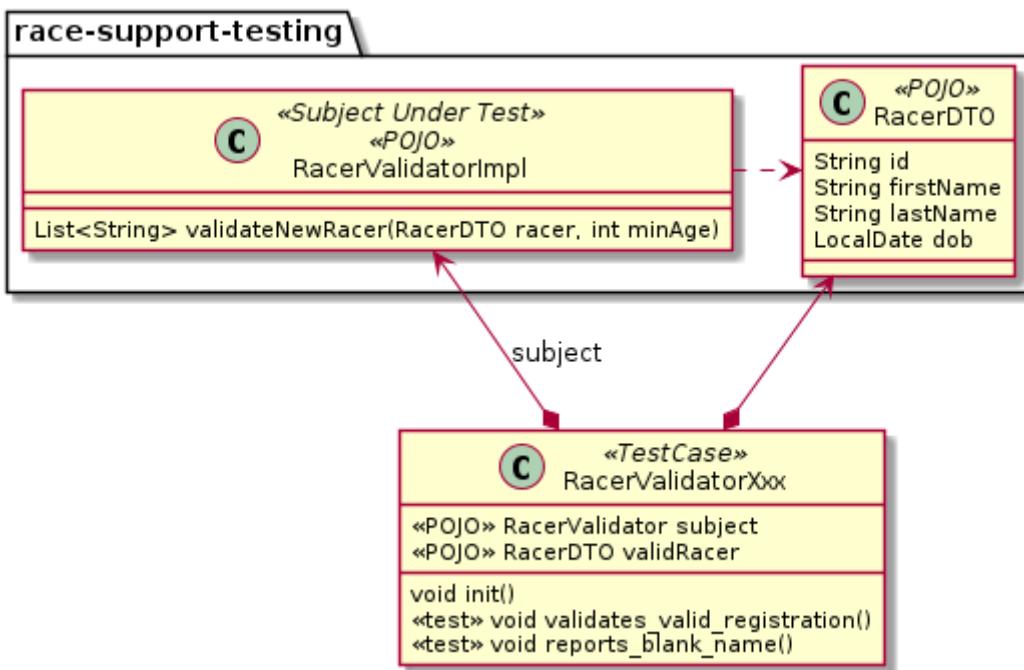


Figure 33. Unit Test

The code under test is 100% complete and provided to you in a separate `race-support-testing` module.

Code Under Test

```
<dependency>
    <groupId>info.ejava.assignments.testing.race</groupId>
    <artifactId>race-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.

119.2.3. Requirements

1. Start with a dependency on supplied and completed `RacerValidatorImpl` and `RacerDTO` classes in the `race-support-impl` module. You only need to understand and test them. You do not need to implement or modify anything being tested.
 - a. `RacerValidatorImpl` implements a `validateNewRacer` method that returns a `List<String>` with identified validation error messages
 - b. `RacerDTO` 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 `RacerValidatorImpl`
 - 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 registration will be reported as valid.
 5. The unit test case must have a test method that verifies an invalid registration 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.

119.2.4. 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 `RaceDTO` 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

119.2.5. Additional Details

1. A quick start project is available in [`assignment-starters/race-starters/assignment1-race-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 `race-support-testing` module. The only things present in your `src/main` would be demonstration code that is supplied to you but not part of

any requirement.

119.3. Mocks

119.3.1. 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

119.3.2. 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.

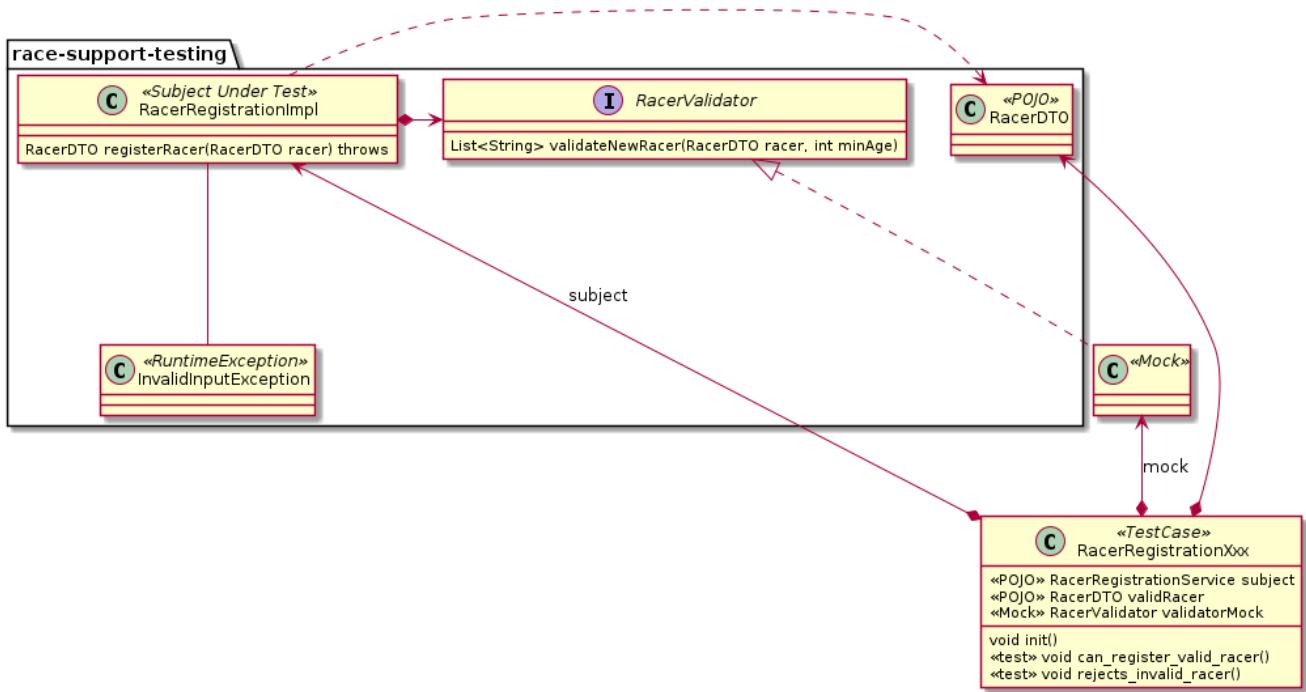


Figure 34. Unit Testing with Mocks

119.3.3. Requirements

1. Start with a dependency on supplied and completed `RacerValidatorImpl` and other classes in the `race-support-impl` module. You only need to understand and test them. You do not need to implement or modify anything being tested.
 - a. `RacerRegistrationService` implements a `registerRacer` method that
 - i. validates the racer using a `RacerValidator` 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 `RacerRegistration` to verify validation for a valid and invalid `RacerDTO`
 - 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 `RacerValidator` instance. You may **not** use the `RacerValidatorImpl` 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 `RacerRegistrationServiceImpl` with the Mock instance for `RacerValidator`.
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 racer.



Understand how the default Mock behaves before going too far with this.

- b. programmatically verify the Mock was called to validate the `RacerDTO` 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 racer
 - b. programmatically verify the Mock was called to validate the `RacerDTO` as part of the test criteria
6. Name the test so that it automatically gets executed by the Maven Surefire plugin.

119.3.4. 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 `RacerValidator` 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 racer
 - b. whether you successfully configured the Mock to return a collection of error messages for the invalid racer
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

119.3.5. 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.

119.4. Mocked Unit Integration Test

119.4.1. 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

119.4.2. 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 `RacerRegistrationServiceImpl` `@Component` for operational functionality and the unit integration test for configuration and inspection commands.

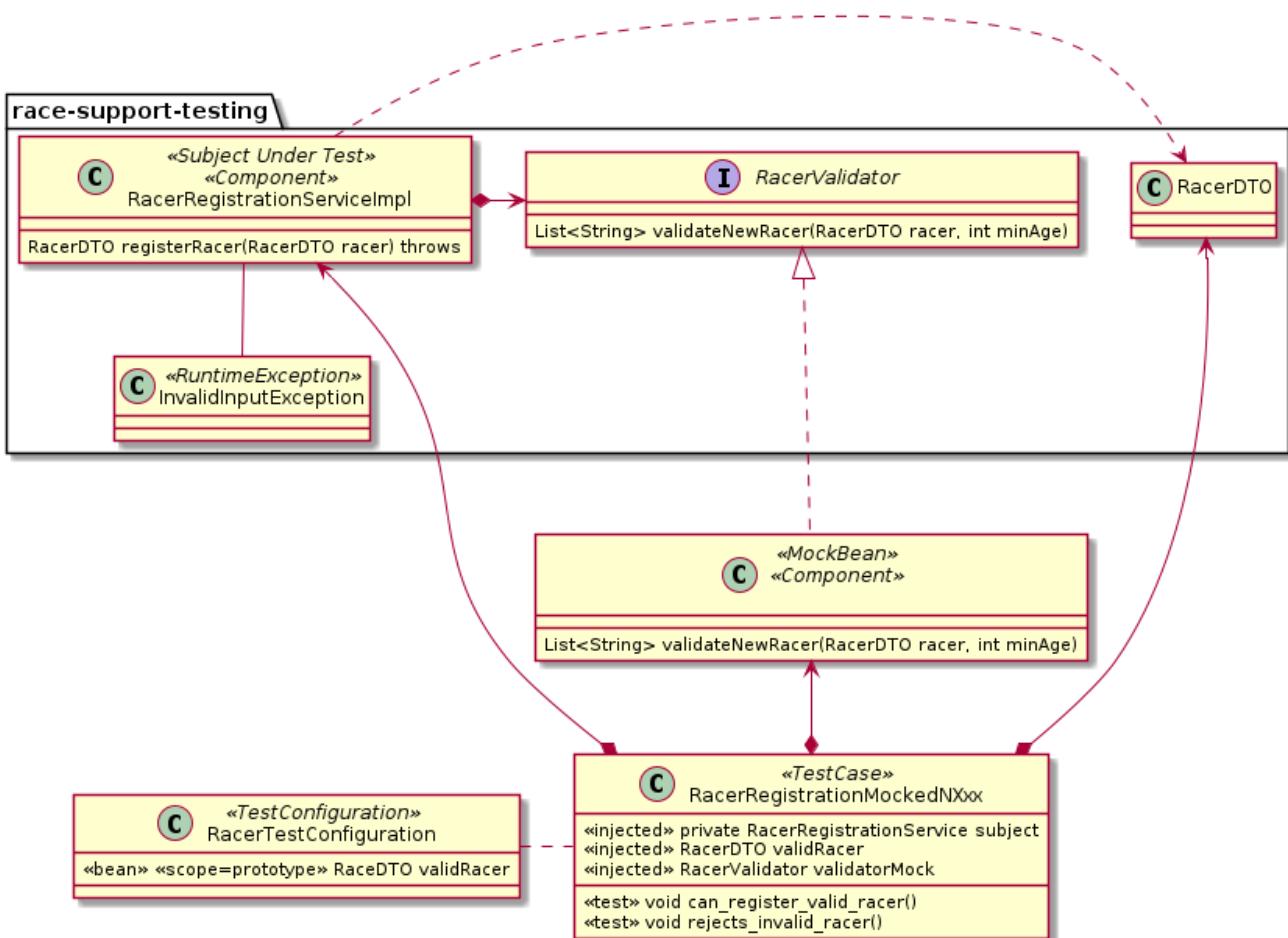


Figure 35. Mocked Unit Integration Test

119.4.3. Requirements

1. Start with a supplied, completed, and injectable 'RacerRegistrationServiceImpl' versus instantiating one like you did in the pure unit tests.
2. Implement a unit integration test for the `RacerRegistration` for a valid and invalid `RacerDTO`
 - 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 `RacerValidator` must be implemented as Mockito Mock/Spring bean and injected into both the `RacerValidatorImpl @Component` and accessible in the unit integration test. You may **not** use the `RacerValidatorImpl` class as part of this test.
 - d. define and inject a `RacerDTO` for a valid racer 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.

119.4.4. 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

119.4.5. 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 `RacerTestConfiguration` to the Spring context using either of the two annotation properties
 - a. `@SpringBootTest.classes`
 - b. `@Import.value`

119.5. Unmocked/BDD Unit Integration Testing

119.5.1. 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

119.5.2. 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.

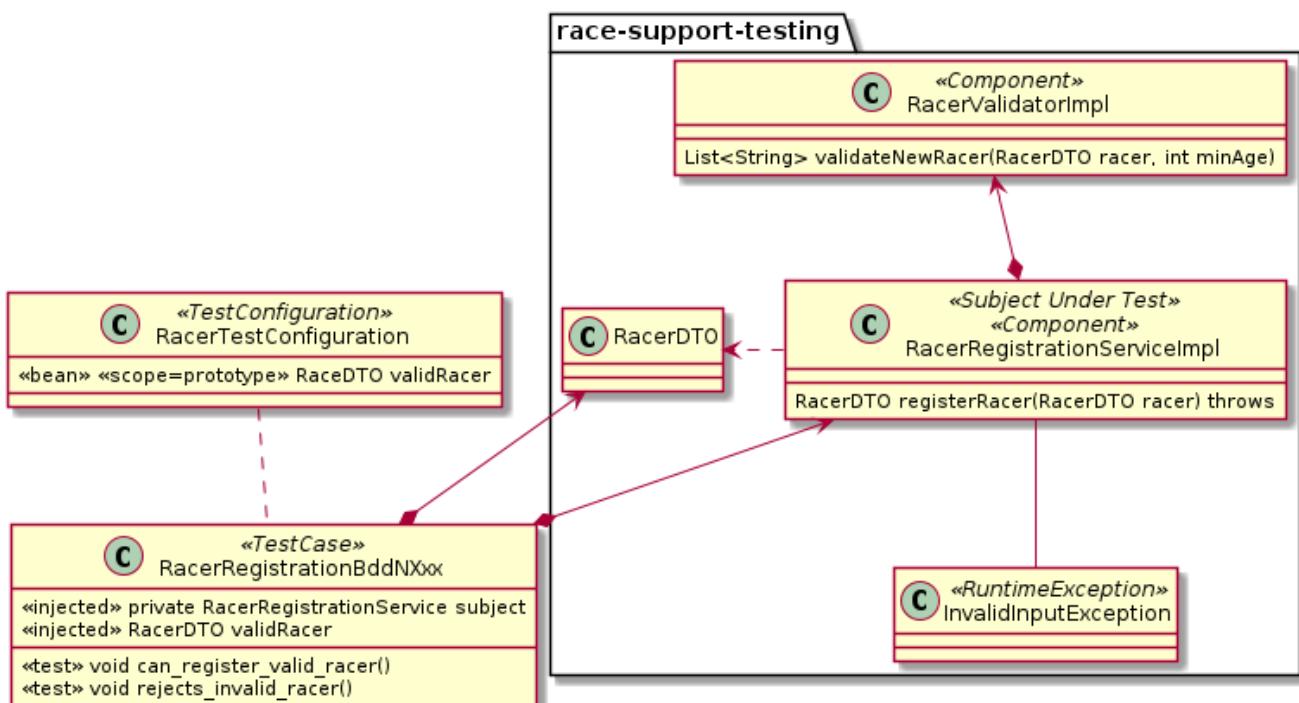


Figure 36. Unmocked/BDD Integration Testing

119.5.3. Requirements

1. Start with a supplied, completed, and injectable `RacerRegistrationServiceImpl` by creating a dependency on the `race-support-testing` module. There are to be no POJOs or Mocks

- implementation of any classes under test — except for the `RaceDTO`.
2. Implement a unit integration test for the `RacerRegistration` for a valid and invalid `RacerDTO`
 - 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 `RacerValidator` must be injected into the `RacerRegistrationServiceImpl` using the Spring context. Your test case will not need access to that component.
 - d. define and inject a `RacerDTO` for a valid racer 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.

119.5.4. 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

119.5.5. 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 names and naming policies for JUnit
 - d. changes AssertJ syntax to BDD form

HTTP API

copyright © 2021 jim stafford (jim.stafford@jhu.edu)

Chapter 120. Introduction

120.1. 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

120.2. 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 121. 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)**.^[18] **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).^[19]

121.1. 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.

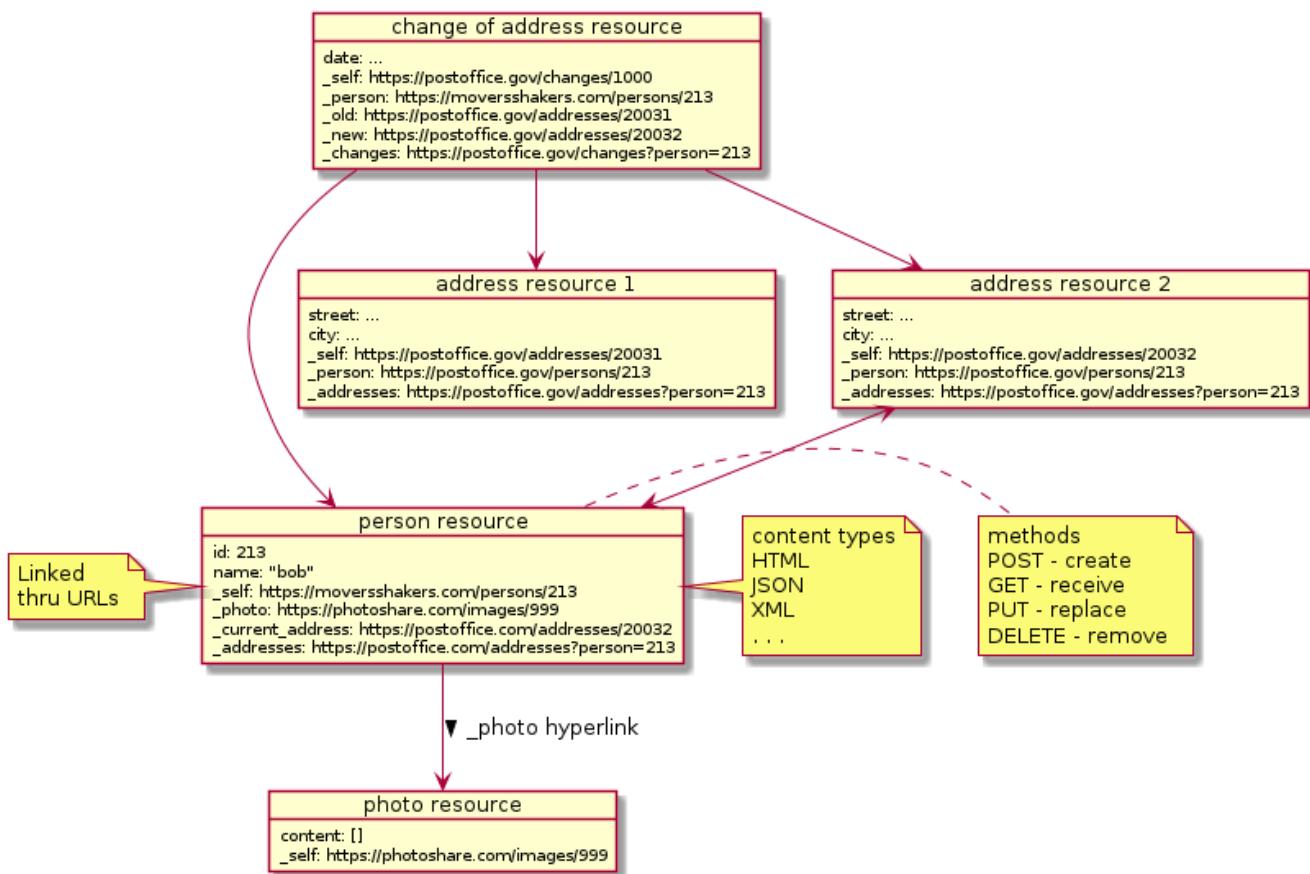


Figure 37. WWW Links Resources thru URLs

[18] "World Wide Web Wikipedia Page"

[19] "Web Resource Wikipedia Page"

Chapter 122. 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 [20]. REST was defined in 2000 by Roy Fielding in his [doctoral dissertation](#) that was also used to design HTTP 1.1. [21] REST relies heavily on the concepts and implementations used in the World Wide Web — which centers around web resources addressable using URIs.

122.1. 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](#)).

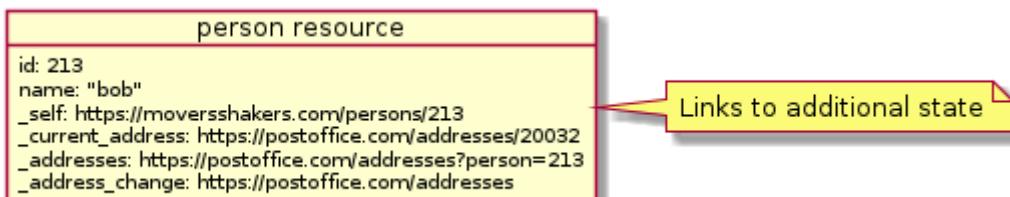


Figure 38. Example of State Represented through Hyperlinks

The interface contract allows clients to dynamically determine current capabilities of a resource and the resource to add capabilities over time.

122.2. Clients Dynamically Discover State

HATEOAS permits the capabilities of client and server to advance independently through the dynamic discovery of links. [22]

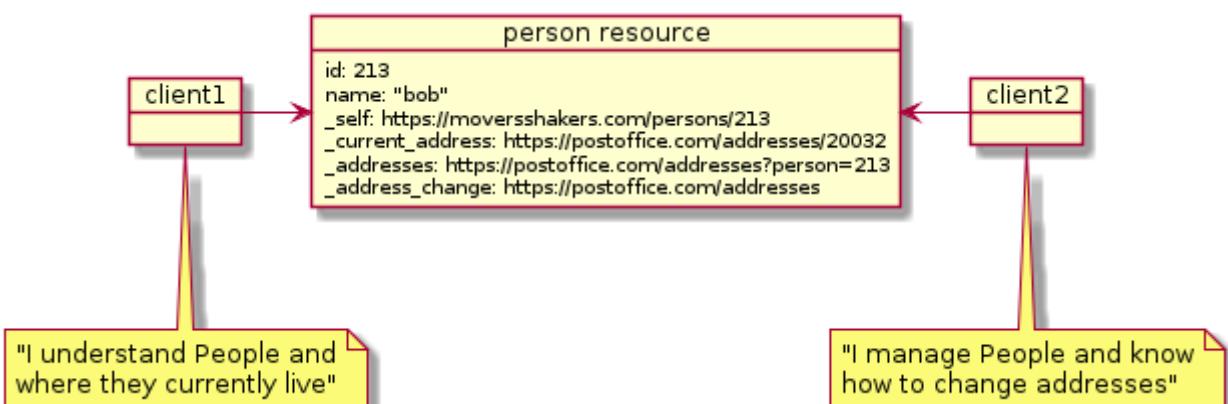


Figure 39. Example of Clients Dynamically Discovering State Represented through Hyperlinks

122.3. 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.

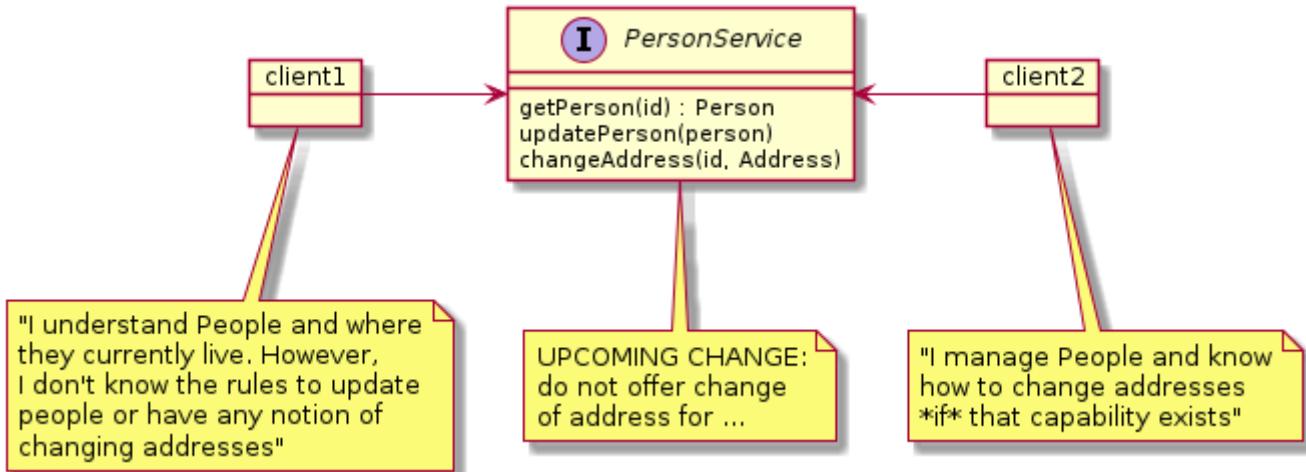


Figure 40. RPC Tight Coupling Example

122.4. 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 provides 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.

122.5. 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?

122.6. Buzzword Association

For many developers and product advertisements eager to get their names associated with a

modern, 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.

122.7. 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).

122.8. Richardson MaturityModel (RMM)

The [Richardson Maturity Model \(RMM\)](#) was developed by Leonard Richardson and breaks down levels of RESTful maturity.^[23] 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 soley as a transport• e.g., CORBA tunneled all calls through HTTP POST

122.9. "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

122.10. 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.

[20] ["Representational state transfer"](#)—Wikipedia Page

[21] ["Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation"](#), Roy Thomas Fielding, University of California, Irvine, 2000

[22] ["HATEOUS Wikipedia Page"](#)

[23] ["Richardson Maturity Model"](#), Martin Fowler, 2010

Chapter 123. 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 124. 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 125. 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

125.1. Nested Resources

Resources can be nested under parent or related resources.

- categories/{id}
- categories/{id}/products
- todos/{name}
- todos/{name}/items

Chapter 126. 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. [\[24\]](#)

— 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 [\[25\]](#)

③ example URN URI; "isbn" is a URN namespace [\[24\]](#)

126.1. 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). [\[24\]](#)

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

126.2. 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:

Generic URI components [26]

```
URI = scheme:[//authority]path[?query][#fragment]
```

The authority component breaks down into subcomponents as follows:

Authority Subcomponents [26]

```
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

126.3. 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
```

126.4. 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(~)
```

126.5. 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](#)

126.6. 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

126.7. 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.

126.8. 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
```

126.9. 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

126.10. 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)  
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
```

[24] "Uniform Resource Identifier (URI): Generic Syntax RFC", Network Working Group, Berners-Lee, Fielding, Masinter, 2005

[25] "The 'mailto' URI Scheme", Duerst, Masinter, Zawinski, 2010

[26] [URI Wikipedia Page](#)

Chapter 127. 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
```

127.1. 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

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 128. 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.

128.1. 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

128.2. 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 129. 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.

129.1. 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 130. 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

130.1. 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 officially 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"
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 131. 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

131.1. 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 132. 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 133. 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 © 2021 jim stafford (jim.stafford@jhu.edu)

Chapter 134. 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.

134.1. 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

134.2. 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 client using Spring MVC RestTemplate
6. implement a client using Spring Webflux in synchronous mode
7. pass parameters between client and service over HTTP
8. return HTTP response details from service
9. access HTTP response details in client

Chapter 135. 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](#).

135.1. Lecture/Course Focus

The focus of this lecture, module, and early portions of the course will be on synchronous communications patterns. The synchronous paradigm is simpler and there are a lot 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 in order to make sense of the software and available documentation. For example, the client-side of Spring MVC (i.e., [RestTemplate](#)) has been deprecated and its duties fulfilled by Spring WebFlux (i.e., [WebClient](#)). Therefore, I will be demonstrating synchronous client concepts using both libraries to help bridge the transition.



WebClient examples demonstrated here are intentionally synchronous

Most early examples of Spring WebFlux's [WebClient](#) will be demonstrated as a synchronous replacement for Spring MVC [RestTemplate](#). The asynchronous, reactive API of [WebClient](#) will be the focus during later asynchronous processing topics.

135.2. 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 have 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.

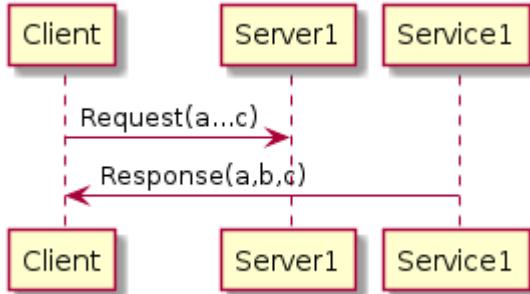


Figure 41. Spring MVC Synchronous Model

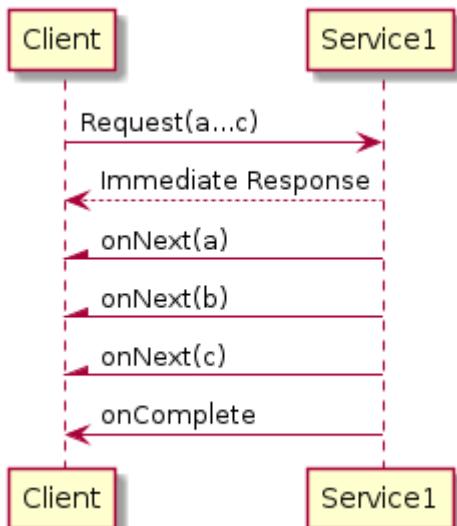
As mentioned earlier, the programming model for Spring MVC is synchronous, blocking request/reply. Each active request is blocked in its own thread while waiting for the result of the current request to complete. This mode scales primarily by adding more threads — most of which are blocked performing some sort of I/O operation.

135.3. 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 are 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 greater concurrent scale.

Figure 42. Spring WebFlux Reactive Model

135.4. 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.

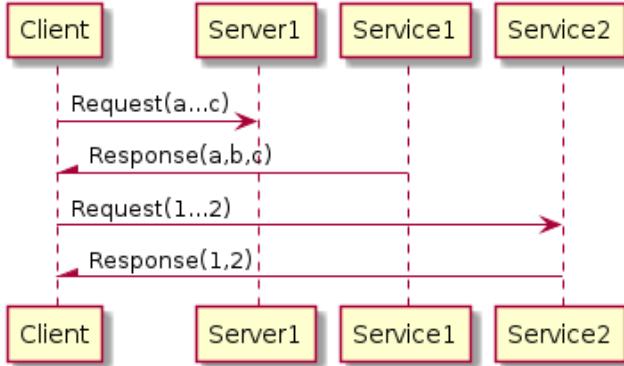


Figure 43. Synchronous

For synchronous, the call to service 2 cannot be initiated until the synchronous call/response from service 1 is completed

For asynchronous, the call to service 1 and 2 are initiated sequentially but are carried out concurrently, and completed independently

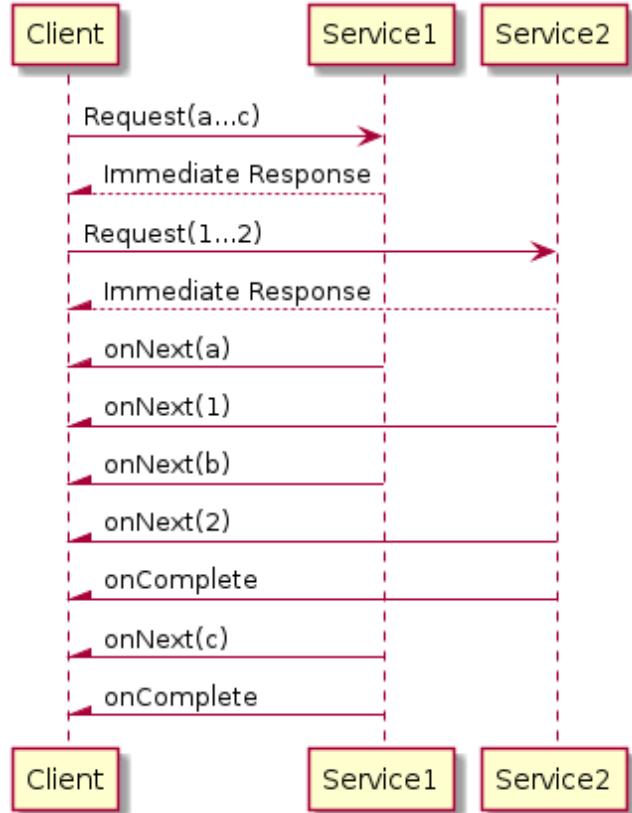


Figure 44. Asynchronous

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 back-pressure of available data and the resources being available to consume it. With the reactive programming paradigm comes strict rules of the road.

135.5. 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, chose between Tomcat or Jetty for the web server, and operate any use of reactive endpoints in a compatibility mode. ^[27]

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.

135.6. Choosing Approaches

Independent synchronous and reactive flows can be formed on a case-by-case basis and optimized if implemented on separate instances.^[27] We can choose our ultimate solution(s) based on some of the recommendations below.

Synchronous

- existing synchronous API working fine—no need to change^[28]
- easier to learn - can use standard Java imperative programming constructs
- easier to debug - everything in same flow is commonly in same thread
- the number of concurrent users is a manageable (e.g., <100) number^[29]
- service is CPU-intensive^[30]
- 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^[29]
- 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^[28]
- service is IO-intensive (e.g., database or external service calls)^[30]

For many of the above reasons, we will start out our HTTP-based API coverage in this course using the synchronous approach.

[27] "Can I use SpringMvc and webflux together?", Brian Clozel, 2018

[28] "Spring WebFlux Documentation - Applicability", version 5.2.6 release

[29] "SpringBoot: Performance War", Santhosh Krishnan, 2020

[30] "Do's and Don'ts: Avoiding First-Time Reactive Programmer Mines", Sergei Egorov, SpringOne Platform, 2019

Chapter 136. 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 137. Sample Application

To get started covering the basics of Web MVC, I am going to use a very simple, 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
  |   |                           '-- RpcGreeterController.java
  |   '-- resources
  |       '-- ...
`-- test
  |-- java
  |   '-- info
  |       '-- ejava
  |           '-- examples
  |               '-- svc
  |                   '-- rpc
  |                       '-- greeter
  |                           |-- GreeterRestTemplateHttpNTest.java
  |                           '-- GreeterSyncWebClientHttpNTest.java
  '-- resources
      '-- ...
```

Chapter 138. 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 {
    //...
}
```

138.1. 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 definition for all methods of the class.

138.2. 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.

138.3. 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

138.4. 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 139. 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. Only the steps directly related to `RestTemplate` are specific to that topic.

139.1. JUnit Integration Test Setup

We start with 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

Example JUnit Integration Unit Test Setup

```
package info.ejava.examples.svc.rpc.greeter;

import info.ejava.examples.svc.rpc.GreeterApplication;
import lombok.extern.slf4j.Slf4j;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.web.server.LocalServerPort;

@SpringBootTest(classes = GreeterApplication.class, ①
               webEnvironment = SpringApplication.WebEnvironment.RANDOM_PORT) ②
@Tag("springboot") @Tag("greeter")
@Slf4j
public class GreeterRestTemplateHttpNTest {
    @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`

139.2. 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.

```

@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

139.3. Obtain RestTemplate

With a URL in hand, we are ready to make the call. We will do that using the synchronous [RestTemplate](#) from the Spring MVC library.

Spring Template 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

```
RestTemplate restTemplate = new RestTemplate();
```

139.4. 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

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, ...

139.5. 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 140. WebClient Client

The Spring 5 documentation states the `RestTemplate` is deprecated and that we should switchover to using the Spring WebFlux `WebClient`. Representatives from Pivotal have stated in various conference talks that `RestTemplate` will likely not go away anytime soon but would likely not get upgrades to any new drivers.

In demonstrating `WebClient`, 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 `WebClient` instance
- invoking the HTTP endpoint endpoint and obtaining result

140.1. 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

```
WebClient webClient = WebClient.builder().build();
```

140.2. Invoke HTTP Call

The methods for `WebClient` 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 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()
    .uri(url)
    .retrieve()
    .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 141. 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

141.1. 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.

141.2. 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` 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 containing the base (e.g. <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

Chapter 142. 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.

142.1. Obtaining ResponseEntity

The two 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 WebClient ResponseEntity<T> Call

```
//when - asking for that greeting
ResponseEntity<String> response = webClient.get()
    .uri(url)
    .retrieve()
    .toEntity(String.class)
    .block();
```

142.2. 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 143. 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—for the most part both `RestTemplate` and `WebClient` will throw an exception if the status code is not successful. Although very similar—unfortunately, their exceptions are technically different and would need separate exception handling logic if used together.

143.1. RestTemplate Response Exceptions

`RestTemplate` is designed to always throw an exception when there is a non-successful status code. Although we can tweak the specific exceptions thrown with filters, we are eventually forced to throw something if we cannot return an object of the requested type or a `ResponseEntity<T>` carrying the requested type.

All default `RestTemplate` 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.getForEntity(url, String.class),
    HttpClientErrorException.BadRequest.class);
```

① using assertj `catchThrowableOfType()` to catch the exception and 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.

143.2. WebClient Response Exceptions

`WebClient` has 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 or a `ResponseEntity<T>` carrying the type requested.

All default `WebClient` exceptions thrown extend `WebClientResponseException`—which is also a `RuntimeException`, so it has that in common with the exception handling of `RestTemplate`. 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);
```

143.3. RestTemplate and WebClient 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`.

Example RestTemplate 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 144. 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

144.1. 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

144.2. Example ResponseEntity Responses

In the 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

144.3. Controller Exception Handler

We can make a small but substantial 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 has the ability to 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) {  
    return ResponseEntity.unprocessableEntity() ②  
        .body(ex.getMessage());  
}
```

① `ExceptionHandler` is registered to handle all `IllegalArgumentException` exceptions thrown by controller method (or anything it calls)

② 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.

144.4. 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 path in order 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 145. 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 how to invoke methods from a Spring MVC `RestTemplate` and Spring WebFlux `WebClient`

Controller/Service Interface

copyright © 2021 jim stafford (jim.stafford@jhu.edu)

Chapter 146. 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 in reality in even a moderately-sized service—there is 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.

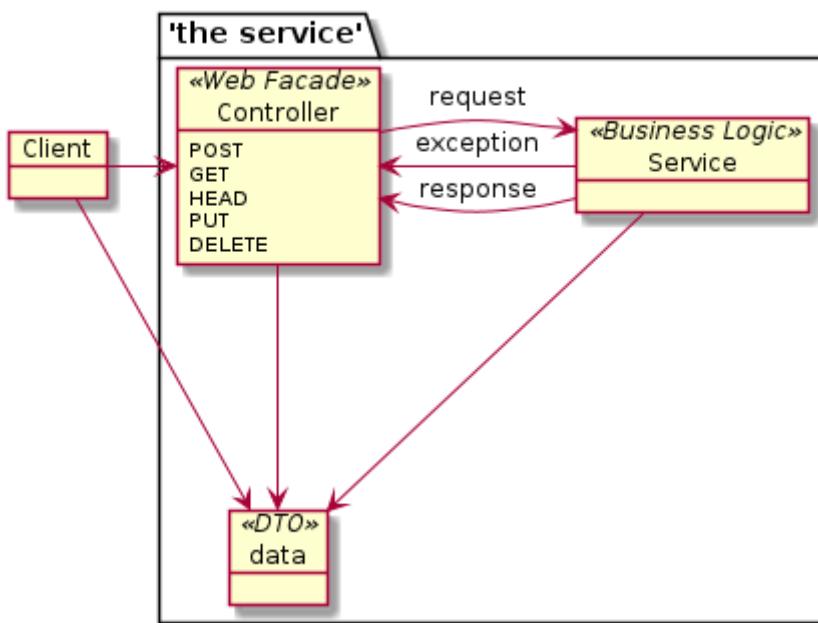


Figure 45. Controller/Service Relationship

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.

146.1. 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

146.2. 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 147. Roles

In an N-tier, distributed architecture there is commonly a set of patterns to apply to our class design.

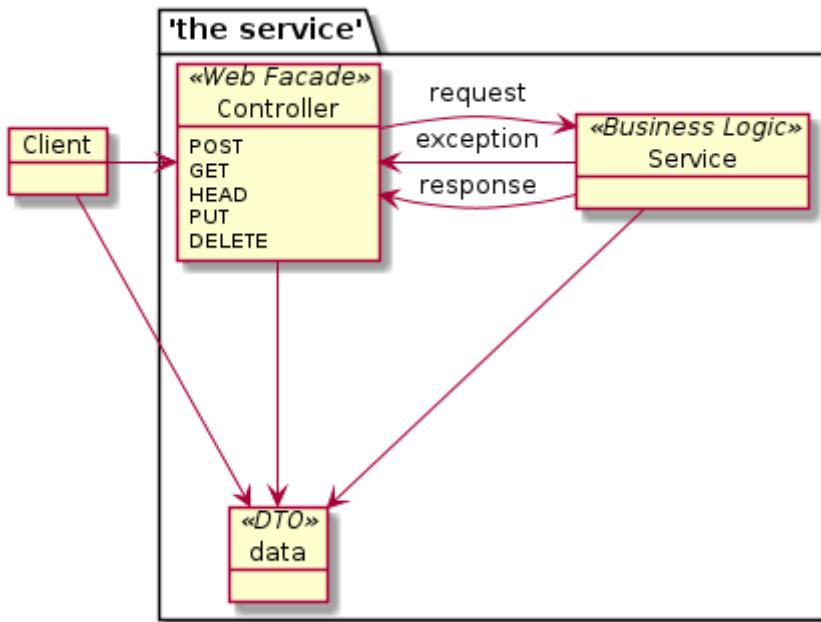


Figure 46. Controller/Service Relationship

- **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 148. 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.

148.1. 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.

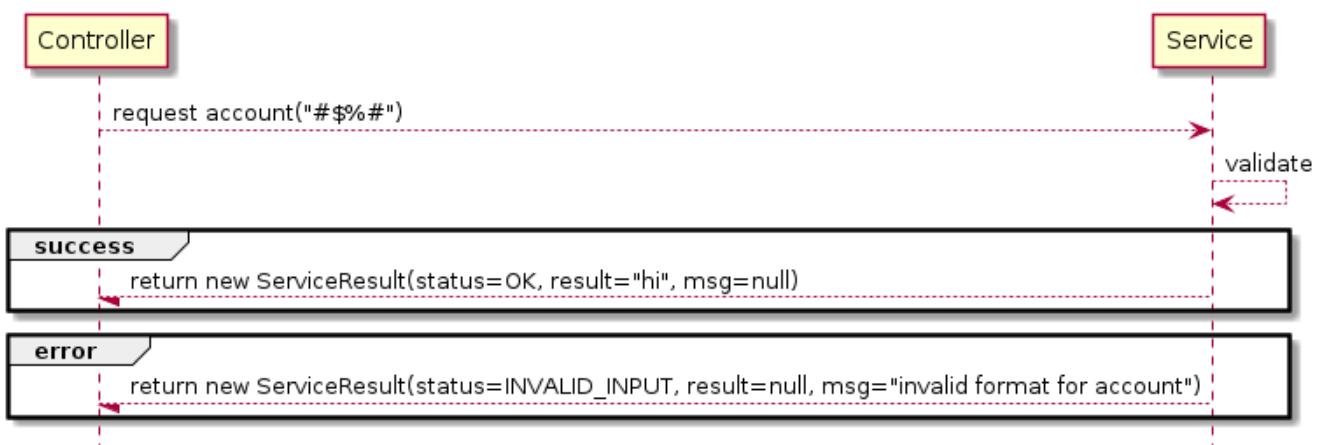


Figure 47. Service Returns Complex Object with Status and Error

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.

148.2. 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.

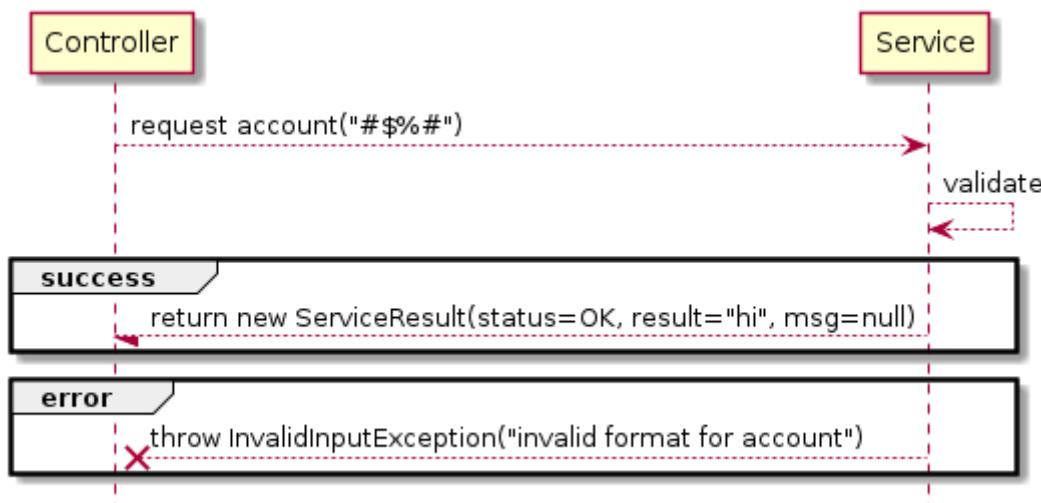


Figure 48. Service Throws Exception of Type of Error

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.

148.3. Exceptions

Going the exception route, we can start to consider

- what specific errors should our services report?
- what information is reported?
- 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.

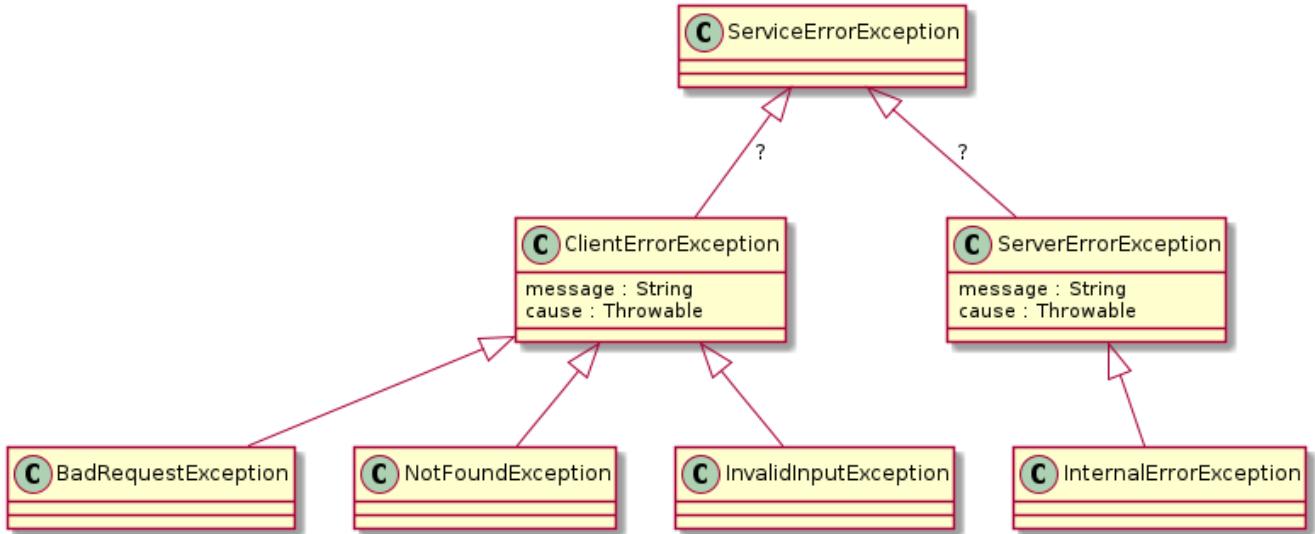


Figure 49. Example Service Exception Hierarchy

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

148.4. 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.

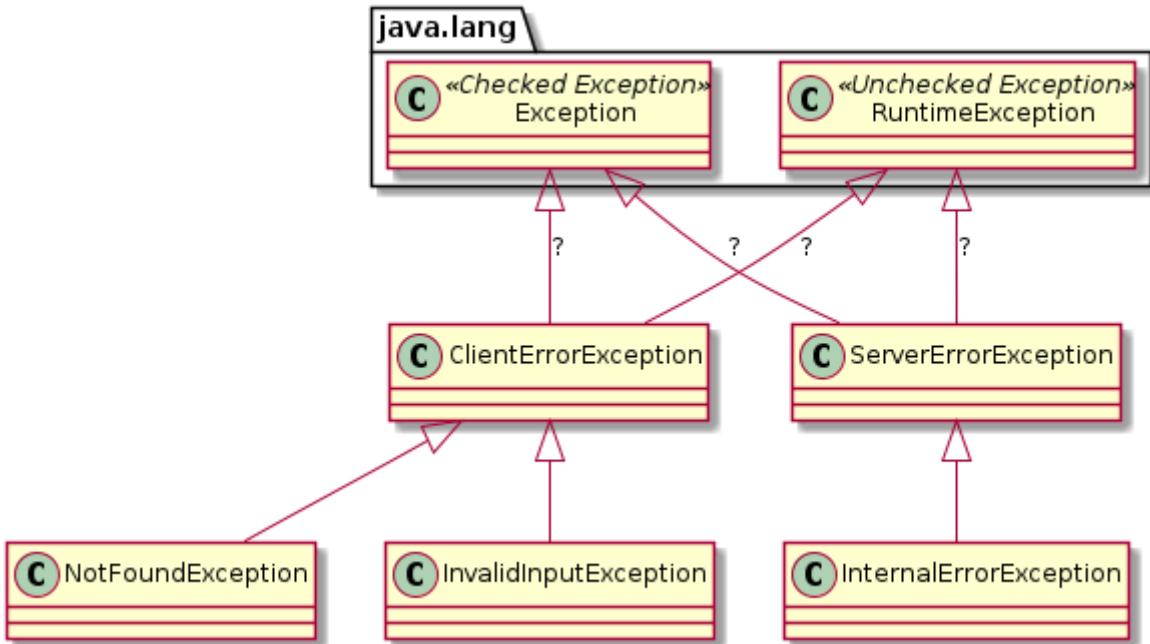


Figure 50. Should Reported Exceptions be Checked or Unchecked?

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).

148.5. 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 (gesture==null) {  
    throw new ClientErrorException  
        .NotFoundException("gesture type[%s] not found", gestureType);  
}
```

148.6. 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 149. 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

149.1. 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 (gesture==null) {
            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

149.2. 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 a 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
    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.

149.3. 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 150. 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 © 2021 jim stafford (jim.stafford@jhu.edu)

Chapter 151. 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 setup properly. Manual marshaling/unmarshalling approaches are mainly useful in determining provider settings and annotations—as well as to perform low-level development debug outside of the server on the shape and content of the payloads.

151.1. 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

151.2. 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 a 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 152. 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.

152.1. 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

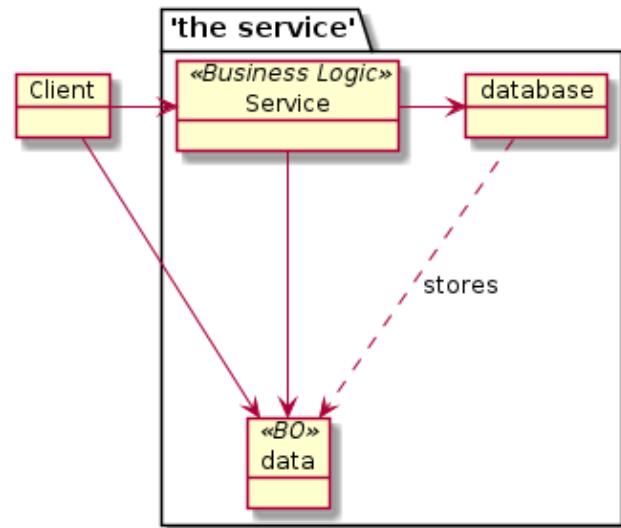


Figure 51. Clients and Service Sharing Implementation Data

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 outside of core scope of business logic

152.2. 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

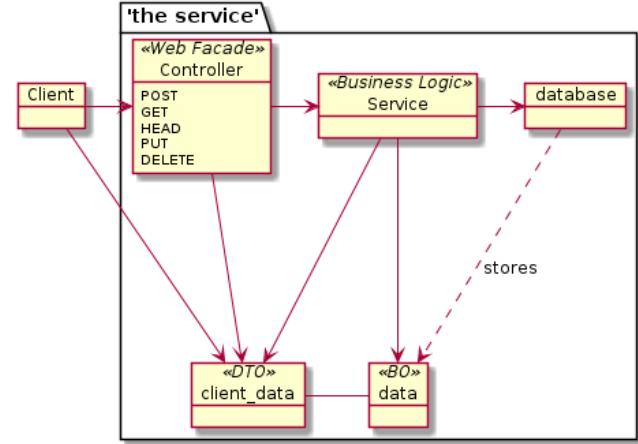


Figure 52. DTO Represents Client View of Data

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.

152.3. 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 even cause small services to split the two data models into separate implementations.

Chapter 153. 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

Chapter 154. 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 birth date, 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?

154.1. Pre Java 8 Time

During pre-Java8, we primarily had the following time-related 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.

154.2. `java.time`

The ambiguity issues with `java.lang.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>OffsetDate</code> <code>Time</code>	adds <code>Calendar</code> -like view to an Instant with a fixed timezone offset.

ZonedDateTime	adds timezone identity to OffsetDateTime — which can be used to determine the appropriate timezone offset (i.e., daylight savings time). 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. A common example of this is a birthday or anniversary. A specific New Year is celebrated at least 24 different times that one day and can be represented with LocalDate .
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.
LocalDateTime	a date and time but lacking a specific timezone offset from UTC. 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.

These are some of the main data classes I will be using in this course. Visit the [javadocs for java.time](#) to see other constructs like [Duration](#), [Period](#), and others.

154.3. 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 java.text 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 DateFormat take care of the details and java.text.SimpleDateFormat accepts a String format specification. An example format <code>yyyy-MM-ddT 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 java.time formatter performs a similar role to DateFormat and SimpleDateFormat combined. It can parse a String into java.time constructs as well as format instances to a String of text. It does not work directly with Dates, but the java.time constructs it does produce can be easily converted to/from java.util.Date thru the Instance type. The coolest thing about DateTimeFormatter 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 ISO_LOCAL_DATE_TIME 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.

154.4. 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 a English word abbreviation for day of week and month. The DateTimeFormatter example for this group is Tue, 3 Jun 2008 11:05:30 GMT ^[31]

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.

[31] "[DateTimeFormatter RFC_1123_DATE_TIME Javadoc](#)", DateTimeFormatter Javadoc, Oracle

Chapter 155. 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, 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 156. 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. Lets 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

```
@NoArgsConstructor  
@NoArgsConstructor  
@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

156.1. 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.

156.1.1. 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 later 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

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 the builder being registered as a `@Bean` factory to be able to adjust Jackson defaults used by the server.

156.1.2. 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

```
public <T> String marshal(T object)
    throws JsonGenerationException, JsonMappingException, IOException {
    StringWriter buffer = new StringWriter();
    mapper.writeValue(buffer, object);
    return buffer.toString();
}
```

The mapper can just as easy — unmarshal the JSON to a POJO instance.

Unmarshal DTO from JSON using Jackson

```
public <T> T unmarshal(Class<T> type, String buffer)
    throws JsonParseException, JsonMappingException, IOException {
    T result = mapper.readValue(buffer, type);
    return result;
}
```

A packaged set of marshal/unmarshal convenience routines have been packaged inside `ejava-dto`

156.1.3. Jackson JSON Maven Aspects

For modules with only DTOs with Jackson annotations, only the 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.

156.2. 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 javax.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

156.2.1. 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 javax.json.bind.Jsonb;  
import javax.json.bind.JsonbBuilder;  
import javax.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

156.2.2. 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

```
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;  
}
```

156.2.3. 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 157. 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

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

157.1. 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

157.1.1. 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 binary 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

157.1.2. 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;
}
```

157.1.3. 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.

```
<dependency>
    <groupId>com.fasterxml.jackson.dataformat</groupId>
    <artifactId>jackson-dataformat-xml</artifactId>
</dependency>
```

157.2. 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 `@XmlElement` annotation with an optional name and namespace defined.

JAXB Requires @XmlElement 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 @XmlElement supplied

```
...  
import javax.xml.bind.annotation.XmlRootElement;  
...  
@XmlElement(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 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 javax.xml.bind.annotation.adapters.XmlAdapter;
...
public static class LocalDateJaxbAdapter extends XmlAdapter<String, LocalDate>
{
    @Override
    public LocalDate unmarshal(String text) {
        return LocalDate.parse(text, DateTimeFormatter.ISO_LOCAL_DATE);
    }
    @Override
    public String marshal(LocalDate timestamp) {
        return 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 javax.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

The alternative is to use a package-level descriptor and have the adapter automatically applied to all properties of that type.

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 javax.xml.bind.annotation.XmlSchema;
import javax.xml.bind.annotation.adapters.XmlJavaTypeAdapter;
import java.time.LocalDate;
```

157.2.1. 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 javax.xml.bind.annotation.XmlSchema;
```

157.2.2. JAXB Marshalling/Unmarshalling

JAXB Imports

```
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Marshaller;
import javax.xml.bind.Unmarshaller;
```

Marshall DTO using JAXB

```
public <T> String marshal(T object) throws JAXBException {
    JAXBContext jbx = JAXBContext.newInstance(object.getClass());
    Marshaller marshaller = jbx.createMarshaller();
    marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true); ①

    StringWriter buffer = new StringWriter();
    marshaller.marshal(object, buffer);
    return buffer.toString();
}
```

① 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;
}
```

157.2.3. JAXB Maven Aspects

Modules that define DTO classes only will require a direct dependency on the `jaxb-api` library for annotations and interfaces.

```
<dependency>
    <groupId>javax.xml.bind</groupId>
    <artifactId>jaxb-api</artifactId>
</dependency>
```

Modules marshalling/unmarshalling DTO classes using JAXB will require a dependency on the following two artifacts. `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 158. Configure Server-side Jackson

158.1. 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>
```

158.2. 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.

Server-side Jackson Builder @Bean Factory

```
...
import com.fasterxml.jackson.databind.SerializationFeature;
import org.springframework.http.converter.json.Jackson2ObjectMapperBuilder;

@SpringBootApplication
public class QuotesApplication {
    public static void main(String...args) {
        SpringApplication.run(QuotesApplication.class, args);
    }

    @Bean
    public Jackson2ObjectMapperBuilder jacksonBuilder() { ①
        return new Jackson2ObjectMapperBuilder()
            .featuresToEnable(SerializationFeature.INDENT_OUTPUT)
            .featuresToDisable(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS);
    }
}
```

① jackson uses `Jackson2ObjectMapperBuilder` instance to create XML and JSON mappers

158.3. 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(QOTE_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) {①
    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);
}
```

① 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 159. Client Marshall Request Content

If we care about the 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 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 160. 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.

160.1. RestTemplate

The following code provides an example of a `RestTemplate` filter that 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 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();
        HttpStatus status = response.getStatusCode();
        String requestBody = new String(body);
        String responseBody = this.readString(response.getBody());
        //... log debug
        return response;
    }
    private String readString(InputStream inputStream) { ... }
    ...
}
```

① RestTemplate gives us access to the client request and response

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
public RestTemplate restTemplate(RestTemplateBuilder builder) {
    RestTemplate restTemplate = builder.requestFactory(
        //used to read the streams twice for logging filter ②
        ()->new BufferingClientHttpRequestFactory(new SimpleClientHttpRequestFactory(
    )))
        .interceptors(List.of(new RestTemplateLoggingFilter())) ①
        .build();
    return restTemplate;
}
```

① the overall intent of this `@Bean` factory is to register the logging filter

② must configure RestTemplate with a buffer for body to enable multiple reads

160.2. 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 161. Date/Time Lenient Parsing and Formatting

In our quote example, we had an easy LocalDateTime 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

Lets 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.

161.1. 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 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] 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 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 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.

161.2. 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

161.3. 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

161.4. 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

161.5. 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 UNMARSHALER = new DateTimeFormatterBuilder()
        // ...
        .toFormatter();
    public static final DateTimeFormatter MARSHALER = 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, UNMARSHALER);
        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);
        MARSHALER.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 UNMARSHALER = 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 Jackson2ObjectMapperBuilder jacksonBuilder() {
    return new Jackson2ObjectMapperBuilder()
        .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 162. 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 © 2021 jim stafford (jim.stafford@jhu.edu)

Chapter 163. 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 access, and distributed application design are core topics of other sibling 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 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 the Swagger Contest Example set of applications in the examples source tree.

163.1. 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

163.2. Objectives

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

1. generate an default Open API 3.0 interface specification using Springfox and 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 Springfox and Springdoc libraries

Chapter 164. 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.

164.1. 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.

164.2. 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.

164.3. 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 year since then. It is not a fast evolving library.

The screenshot shows a browser window displaying the Springfox Swagger UI at <http://localhost:8080/swagger-ui/index.html#/quotes-controller>. The page title is "Api Documentation 1.0 OAS3". Below the title, there are links for "Terms of service" and "Apache 2.0". A "Servers" dropdown is set to "http://localhost:8080 - Inferred Url". The main content area lists API endpoints under the "quotes-controller" section. Each endpoint is represented by a colored button indicating its HTTP method and path. The endpoints listed are:

- GET /api/quotes getQuotes
- POST /api/quotes createQuote
- DELETE /api/quotes deleteAllQuotes
- GET /api/quotes/{id} getQuote
- PUT /api/quotes/{id} updateQuote
- DELETE /api/quotes/{id} deleteQuote
- HEAD /api/quotes/{id} containsQuote
- GET /api/quotes/random randomQuote

Below the "quotes-controller" section, there is another section for "web-mvc-links-handler".

Figure 53. Example Springfox Swagger UI

164.4. 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. That indicates to me that they have a lot of catch-up to do to complete the product. However, they have the advantage of coming in when the standard is 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 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 'GARD' 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. To the right of the endpoints, there's a 'Schemas' section showing 'QuoteDTO' and 'QuoteListDTO' with their respective descriptions and examples.

Figure 54. Example Springdoc SwaggerUI

Chapter 165. 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.

165.1. Springfox Minimal Configuration

Springfox requires one change to a web application to support Open API 3 and the SwaggerUI:

- add Maven dependencies

165.1.1. Springfox Maven POM Dependency

Springfox requires two dependencies—which are both automatically brought in by the following starter dependency.

Springfox Maven POM Dependency

```
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-boot-starter</artifactId>
</dependency>
```

The starter automatically brings in the following dependencies—that no longer need to be explicitly named.

springfox-boot-starter Dependencies

```
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId> ①
</dependency>
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId> ②
</dependency>
```

① support for generating the Open API spec

② support for the Swagger UI

If you are implementing a module with only the DTOs or controllers and working to further define the API with annotations, you would only need the [springfox-swagger2](#) dependency.

165.1.2. Springfox Access

Once that is in place, you can access

- Open API spec: <http://localhost:8080/v2/api-docs>
- Swagger UI: <http://localhost:8080/swagger-ui/index.html>

The minimally configured Springfox will display more than what we want—but it has what we want.

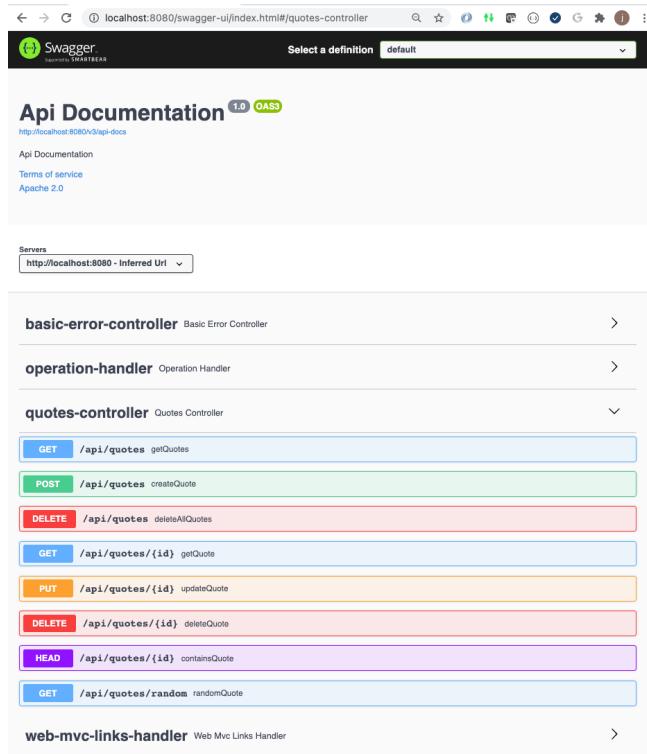


Figure 55. Minimally Configured SpringFox

165.1.3. Springfox Starting Customization

The starting point for adjusting the overall interface is done thru the definition of one or more Dockets. From here, we can control the path and [dozens of other options](#). The specific option shown will reduce the operations shown to those that have paths that start with "/api/".

Springfox Starting Customization

```
import springfox.documentation.builders.PathSelectors;
import springfox.documentation.spi.DocumentationType;
import springfox.documentation.spring.web.plugins.Docket;

@Configuration
public class SwaggerConfig {
    @Bean
    public Docket api() {
        return new Docket(DocumentationType.SWAGGER_2)
            .select()
            .paths(PathSelectors.regex("/api/.+"))
            .build();
    }
}
```

Textual descriptions are primarily added to the annotations of each controller and model/DTO class.

165.2. Springdoc Minimal Configuration

Springdoc minimal configuration is as simple as it gets. All that is required is a single Maven dependency.

165.2.1. Springdoc Maven Dependency

Springdoc has a single top-level dependency that brings in many lower-level dependencies.

Springdoc Maven Dependency

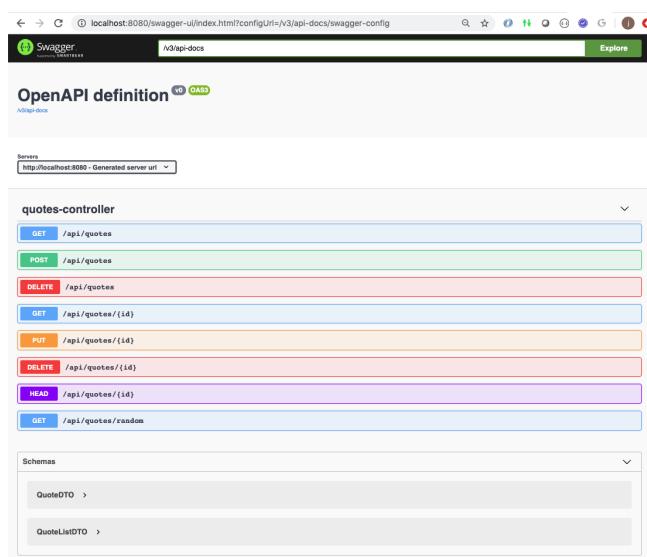
```
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-ui</artifactId>
</dependency>
```

165.2.2. 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 filters out some of the Springboot overhead APIs and what we get is more tuned towards our developed API.



Avoid the Petstore



The `/swagger-ui.html` URI gets redirected to `/swagger-ui/index.html?configUrl=v3/api-docs/swagger-config` when called. The `/swagger-ui/index.html` URI defaults to the Petstore application if `configUrl` is not supplied. Be sure to use the `/swagger-ui.html` URI when trying to serve up the current application or supply the `configUrl` parameter.

165.2.3. Springdoc Starting Customization

The starting point for adjusting the overall interface for Springdoc is done thru 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.GroupedOpenApi;
import org.springdoc.core.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 166. Example Use

By this point in time we are past-ready for a live demo. You are invited to start both the Springfox and 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 Springfox Demo App

```
$ mvn spring-boot:run -f springfox-contest-svc \①  
-Dspring-boot.run.arguments==server.port=8081 ②
```

① starts the web application from within Maven

② passes arguments from command line, thru Maven, to the Spring Boot application

Start Springdoc Demo App

```
$ mvn spring-boot:run -f springdoc-contest-svc \  
-Dspring-boot.run.arguments==server.port=8082 ①
```

① using a different port number to be able to compare side-by-side

Access the two versions of the application using

- Springfox: <http://localhost:8081/swagger-ui/index.html>
- Springdoc: <http://localhost:8082/swagger-ui.html>

I will show an example thread here that is common to both.

166.1. Access Contest Controller POST Command

1. click on the **POST /api/contests** line and the details of the operation will be displayed.
2. select a content type (**application/json** or **application/xml**)
3. click on the "Try it out" button.

The screenshot shows the Swagger UI interface for the **contest-controller** API. The **POST /api/contests** endpoint is selected. The **Try it out** button is highlighted with a red circle. Below it, the **Parameters** section shows a sample JSON object for **newContest**. The **Responses** section displays a curl command and the resulting XML response for status code 201.

```

curl -X POST "http://localhost:8081/api/contests" -H "Content-Type: application/xml" -d "<?xml version='1.0'?><contest xmlns='urn:schemas.svc-messenger.contests' id='21'><scheduledStart xmlns='http://schemas.microsoft.com/2003/10/Serialization/DateTimeOffset'>2020-06-16T21:01:39.935Z</scheduledStart><duration>PT60M</duration><completed>true</completed><homeScore>2</homeScore><awayScore>1</awayScore></contest>" --output /tmp/test.xml
  
```

```

<?xml version="1.0"?>
<contest xmlns="urn:schemas.svc-messenger.contests" id="21">
  <scheduledStart xmlns="">2020-06-16T21:01:39.935Z</scheduledStart>
  <duration>PT60M</duration>
  <completed>true</completed>
  <homeScore>2</homeScore>
  <awayScore>1</awayScore>
</contest>
  
```

166.2. Invoke Contest Controller POST Command

1. Overwrite the values of the example with your values.
2. Select the desired response content type (**application/json** or **application/xml**)
3. press "Execute" to invoke the command

The screenshot shows the Swagger UI interface for the **contest-controller** API. The **POST /api/contests** endpoint is selected. The **Execute** button is highlighted with a red circle. Below it, the **Parameters** section shows a sample JSON object for **newContest**.

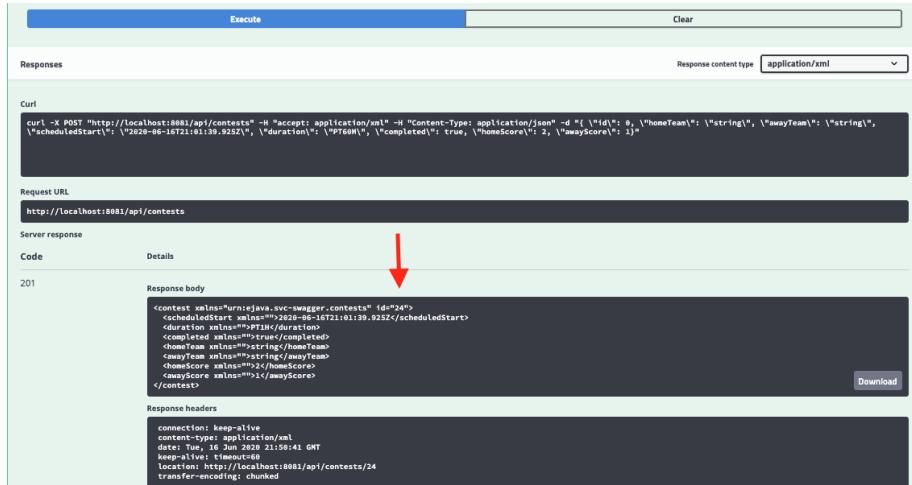
166.3. View Contest Controller POST Command Results

1. look below the "Execute" button to view the results of the command. There will be a payload and some response headers.

2. notice the payload returned will have an ID assigned

3. notice the headers returned will have a location header with a URL to the created contest

4. 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.



The screenshot shows a REST API tool interface. At the top, there's a blue bar with 'Execute' and 'Clear' buttons. Below it, a 'Responses' section has 'application/xml' selected as the response content type. The 'Curl' section shows a POST command to 'http://localhost:8081/api/contests'. The 'Request URL' is 'http://localhost:8081/api/contests'. The 'Server response' section shows a 201 status code. The 'Details' tab is selected, displaying the 'Response body' which is an XML document with contest details. A red arrow points down to the 'Response headers' section, which includes 'connection: keep-alive', 'content-type: application/xml', 'date: Mon, 20 Apr 2020 21:58:41 GMT', 'keep-alive: timeout=5', 'location: http://localhost:8081/api/contests/24', and 'transfer-encoding: chunked'.

```

<?xml version="1.0" encoding="UTF-8"?>
<contest xmlns="urn:java:svc-mugger.contexts" id="24">
    <scheduledStart>2020-04-16T18:39:32Z</scheduledStart>
    <duration>PT1H</duration>
    <completed>true</completed>
    <homeTeam>Seattle Sounders</homeTeam>
    <awayTeam>New York City FC</awayTeam>
    <homeScore>2</homeScore>
    <awayScore>1</awayScore>
</contest>
  
```

Download

connection: keep-alive
 content-type: application/xml
 date: Mon, 20 Apr 2020 21:58:41 GMT
 keep-alive: timeout=5
 location: http://localhost:8081/api/contests/24
 transfer-encoding: chunked

Chapter 167. 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. for both Springfox and Springdoc to get a sense of how they performed.

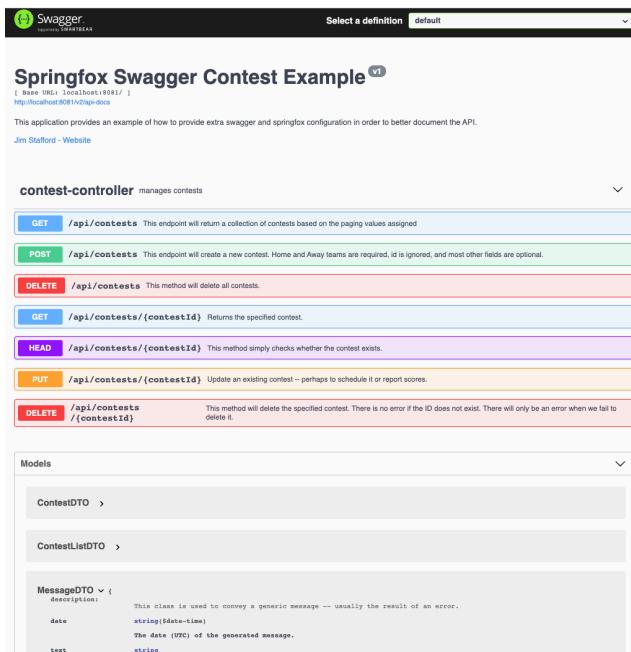


Figure 56. Fully Configured Springfox Example

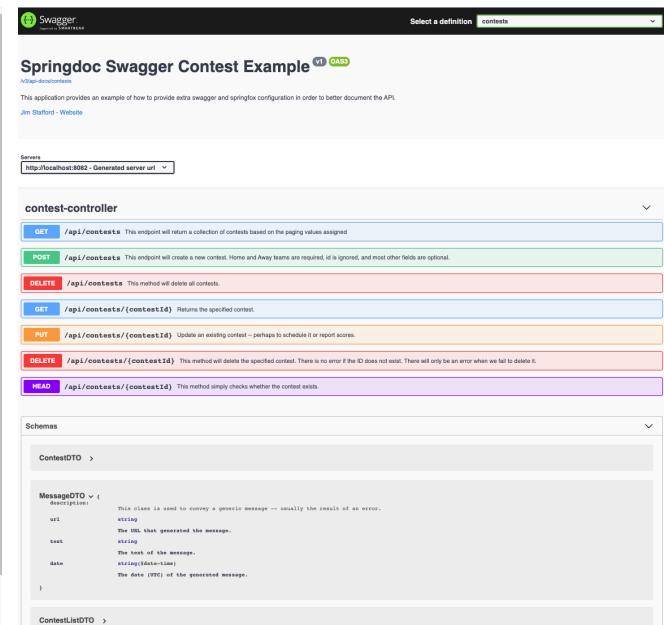


Figure 57. Fully Configured Springdoc Example

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 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.

167.1. Customizing Type Expressions

`java.time.Duration` has a simple ISO string format expression that looks like `PT60M` or `PT3H` for periods of time.

167.1.1. OpenAPI 2 Model Property Annotations

The following snippet shows the Duration property enhanced with Open API 2 annotations to use a default **PT60M** example value.

ContestDTO Snippet with Duration and OpenAPI 2 Annotations

```
package info.ejava.examples.svc.springfox.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.annotations.ApiModel;
import io.swagger.annotations.ApiModelProperty;

@JacksonXmlRootElement(localName="contest", namespace=ContestDTO.CONTEST_NAMESPACE)
@ApiModel(description="This class describes a contest between a home and, " +
    " away team, either in the past or future.")
public class ContestDTO {
    @JsonProperty(required = false)
    @ApiModelProperty(position = 4,
        example = "PT60M",
        value="Each scheduled contest should have a period of time specified " +
            "that identifies the duration of the contest. e.g., PT60M, PT2H")
    private Duration duration;
```

167.1.2. OpenAPI 3 Model Property Annotations

The following snippet shows the Duration property enhanced with Open API 3 annotations to use a default **PT60M** example value.

ContestDTO Snippet with Duration and OpenAPI 3 Annotations

```
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(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;
```

167.2. Duration Example Renderings

Both Springfox and Springdoc derive a more complicated schema either for JSON, XML, or both that was desired or usable.

Springfox has a complex definition for `java.util.Duration` for both JSON and XML.

Springfox Default Duration JSON Expression

```
"duration": {  
    "nano": 0,  
    "negative": true,  
    "seconds": 0,  
    "units": [  
        {  
            "dateBased": true,  
            "durationEstimated": true,  
            "timeBased": true  
        }  
    ],  
    "zero": true  
},
```

Springfox Default Duration XML Expression

```
<duration>  
    <nano>0</nano>  
    <negative>true</negative>  
    <seconds>0</seconds>  
    <units>  
        <durationEstimated>  
        true</durationEstimated>  
        <timeBased>true</timeBased>  
    </units>  
    <zero>true</zero>  
</duration>
```

Springdoc has a fine default for JSON but a similar issue for XML.

Springdoc Default Duration JSON Expression

```
"duration": "PT60M",
```

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>
```

We can correct the problem in Springfox by mapping the Duration class to a String. I originally found this solution for one of the other `java.time` types and it worked here as well.

Example Springfox Map Class to Alternate Type

```
@Bean
public Docket api(SwaggerConfiguration config) {
    return new Docket(DocumentationType.SWAGGER_2)
        .select()
        .paths(PathSelectors.regex("/api/.*"))
        .build()
        .directModelSubstitute(Duration.class, String.class)
        // ...
    ;
}
```

With the above configuration in place, Springfox provides an example that uses a simple string to express ISO duration values.

Springfox Duration Mapped to String JSON Expression

```
"duration": "PT60M",
```

Springfox Duration Mapped to String XML Expression

```
<duration>PT60M</duration>
```

Judging by the fact that Springdoc is new—post Java 8 and expresses a Duration as a string for JSON, tells me there has to be a good solution for the XML side. I did not have the time to get a perfect solution, but found a configuration option that at least expressed the Duration as an empty string that was easy to enter in a value.

Example Springdoc Map Class to Alternate Type

```
@Bean
public GroupedOpenApi api(SwaggerConfiguration config) {
    SpringDocUtils.getConfig()
        .replaceWithSchema(Duration.class,
            new Schema().example("PT120M")
        );
    return GroupedOpenApi.builder()
        .group("contests")
        .pathsToMatch("/api/contests/**")
        .build();
}
```

The examples below shows the configuration above improved the XML example without breaking the JSON example that we were targeting from the beginning. I purposely chose an alternate Duration value so we could see that the global configuration for property types is overriding the individual annotations.

Springfox Duration Mapped to String JSON Expression

```
"duration": "PT120M",
```

Springfox Duration Mapped to String XML Expression

```
<duration>  
</duration>
```

Chapter 168. Springfox / Springdoc Analysis

Both of these packages are surprisingly functional right out of the box with the minimal configuration—with the exception of some complex types. In early June 2020, Springdoc definitely understood the purpose of the Java code better than Springfox. That is likely because Springdoc is very much aware of Spring Boot 2.x and Springfox is slow to evolve.

The one feature I could not get to work in either—that I assume works—is "examples" for complex types. I worked until I got a legal JSON and XML example displayed but fell short of being able to supply an example that was meaningful to the problem domain (e.g., supplying team names versus "string"). 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.

(In early June 2020) Springfox had better documentation that shows you features ahead of time in logical order. Springdoc's documentation was primarily a Q&A FAQ that showed features in random order. I could not locate a good Springdoc example—but after implementing with Springfox first, the translation was extremely easy.

Springfox has 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 will likely not be a good choice for Open API or Webflux users. Springdoc seems like it is having some learning pains—where features may work easier but don't always work 100%, lack of documentation and examples to help correct, and their existing FAQ samples do not always match the code. However, it seems solid already (in early June 2020) for our purpose and they are issuing many releases per month since they first commit in July 2019. By the time you read this much will 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 names of controllers, models/DTO classes, parameters, and properties 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 169. 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 — are focused on implementing Open API for Spring and Spring Boot applications and provide 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 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, but now supports Open API 3 and WebFlux
- learned that Springdoc is newer, active, and supports Open API 3, SpringMVC, and Webflux
- learned how to minimally configure Springfox and Springdoc into our web application in order to provide the simple ability to invoke our HTTP endpoint operations.

Assignment 2 API

copyright © 2021 jim stafford (jim.stafford@jhu.edu)

The parts of the API assignment make up a single assignment that is broken into focus areas that relate 1:1 with the lecture topics. You have the individual choice to start with any one area and either advance or jump repeatedly between them as you complete it as one overall solution. However, you are likely going to want to start out with 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 170. Overview

The API will include three main concepts. We are going to try to keep the business rules pretty simple at this point:

1. **Race** - an individual race for a specific time and place
 - a. Races can be created, modified, listed, and deleted
 - b. Races can be deleted entirely at any time.
2. **Racer** - identification for a person that may register for races and is not specific to any one race
 - a. Racer information can be created, modified, listed, and deleted
 - b. Racer information can be modified or deleted at any time
3. **RaceRegistration** - registration of one racer to a race
 - a. RacerRegistrations can be created for an existing race and racer
 - b. Any racer information pertinent to race will be locked into this registration at creation time
 - c. Registrations cannot be modified but they can be deleted and re-created



Modifications are Replacements

All modifications are replacements. There are no individual field edits requested.

170.1. Grading Emphasis

Grading emphasis will be focused on the demonstration of satisfaction of the listed learning objectives and 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 the individually stated requirements and completing the listed scenarios at the end of the assignment as the minimal functionality required.

170.2. Race Support

You are given a complete implementation of Races and Racers as examples and building blocks in order to complete the assignment. Your primary work will be in completing the RaceRegistrations.

The `races-support-api-svc` module contains a full @RestController/Service/Repo thread for both Races and Racers. The module contains two Auto-configuration definitions that will automatically activate the two services within any containing web application.

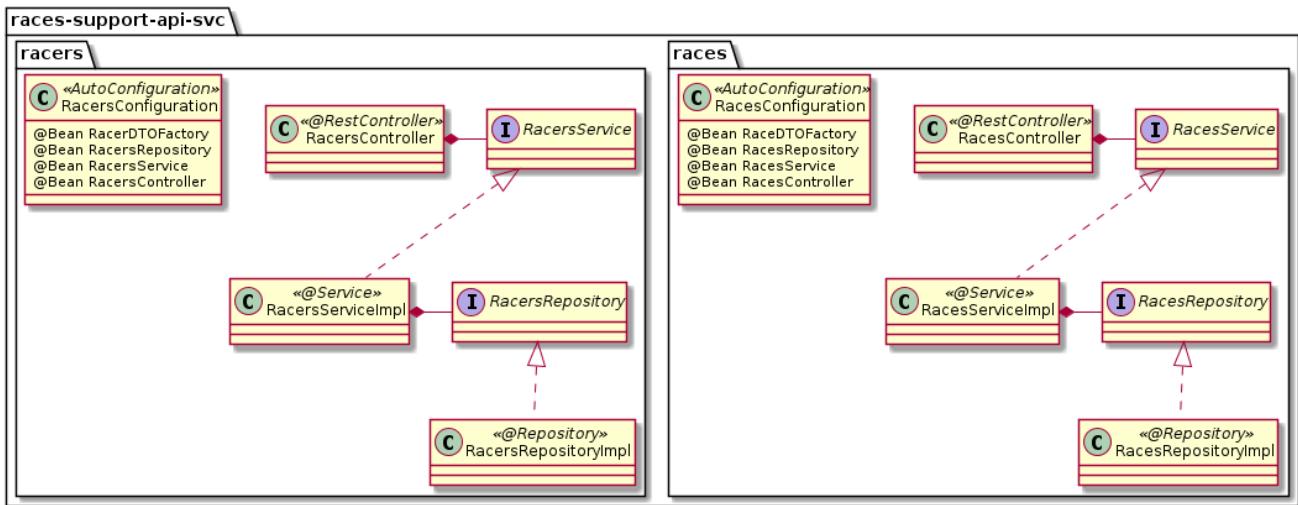


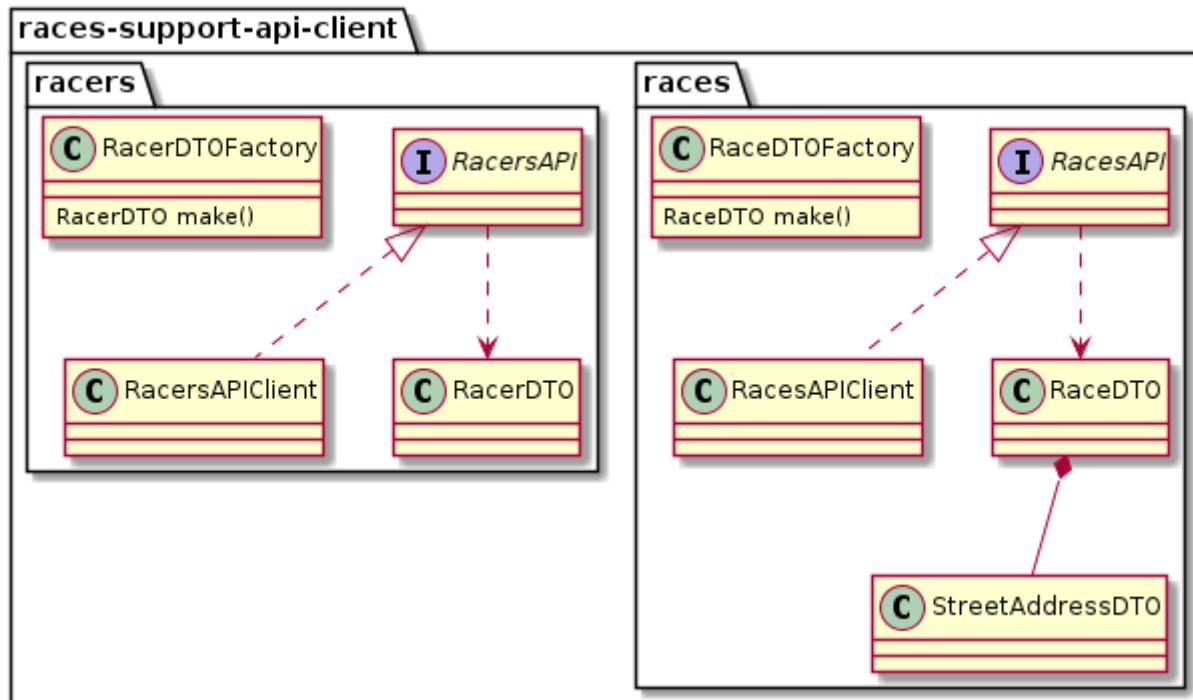
Figure 58. races-support-api-svc Module

The following dependency can be added to your solution to bring in the Races and Racers service examples to mimic and build upon.

race-support-api-svc Dependency

```
<dependency>
    <groupId>info.ejava.assignments.api.race</groupId>
    <artifactId>race-support-api-svc</artifactId>
    <version>${ejava.version}</version>
</dependency>
```

A client module is supplied that includes the DTOs and client to conveniently communicate with the APIs. Your RaceRegistration solution may inject the Races and Racers service components for interaction but your API tests will need to use the Race and Races API.



The following dependency can be added to your solution to bring in the Races and Racers client

artifact examples to mimic and build upon.

race-support-api-client Dependency

```
<dependency>
    <groupId>info.ejava.assignments.api.race</groupId>
    <artifactId>race-support-api-client</artifactId>
    <version>${ejava.version}</version>
</dependency>
```

Chapter 171. Assignment 2a: Modules

171.1. 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

171.2. 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 modules should declare a dependency on either of the provided `race-support-api-client` and/or `race-support-api-svc` modules as needed. You do not copy or clone these. Create a dependency on these and use them as delivered.

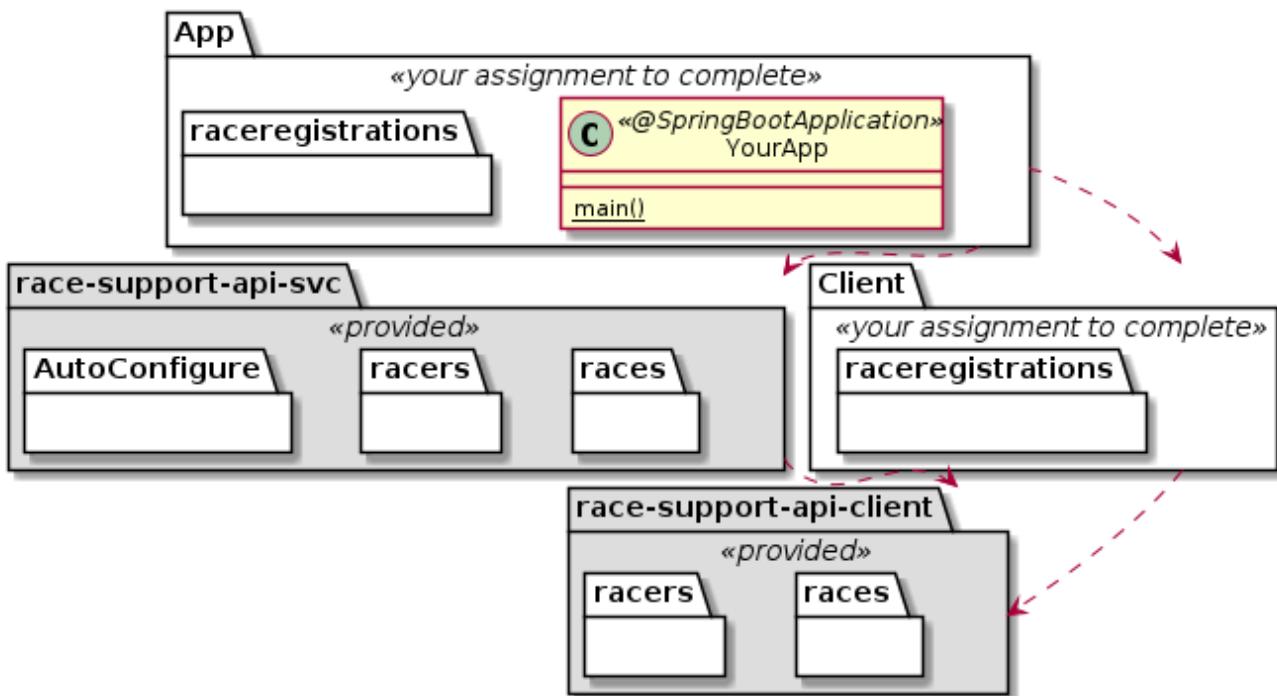


Figure 59. Module Packaging

171.3. 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. `race-support-api-client` has been supplied

for you use as an example and part of your client modules. Create a dependency on this module. Do not copy/clone it unless you desire to do a lot of unnecessary extra work.

- b. **app** module that contains the `@SpringBootApplication` class and can optionally produce a Spring Boot Executable JAR. It will also include your `RaceRegistration` controller, service, and repository work. `race-support-api-svc` has been supplied for you to use as an example and part of your service modules. Create a dependency on this module. Do not copy/clone it unless you desire to do a lot of unnecessary extra work.



The Maven `pom.xml` in the assignment starter for the App uses a profile to build a standard library JAR by default and an executable JAR by profile option. By following this approach, you can make this assignment immediately reusable in assignment 3.

- c. **parent** module that establishes a common `groupId` and version for the child modules and delegate build commands. Only your app and client modules will be children of this parent. The `race-support-api-client` and `race-support-api-svc` modules are embedded within the class example tree and must be accessed from the repository.
2. Add a `@SpringBootApplication` class to the app module (already provided in starter for initial demo).
 3. Define the module as a Web Application (dependency on `spring-boot-starter-web`).
 4. Once constructed, the modules should be able to
 - a. build the project from the root level
 - b. build regular Java JARs for the non-application (client) modules for use in other applications
 - c. (optionally) build one Spring Boot executable JAR for the application with a `@SpringBootApplication` class
 - d. immediately be able to access the `/api/races` and `/api/racers` resource API when the application is running—because of Auto-Configuration.

Example Calls to Races and Racers Resource API

```
$ curl -X GET http://localhost:8080/api/races
{"races":[]}
$ curl -X GET http://localhost:8080/api/racers
{"racers":[]}
```

171.4. 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 Races and Racers API

171.5. Additional Details

1. Pick a Maven hierarchical groupId for your modules that is unique to your overall work on this assignment.

Chapter 172. Assignment 2b: Content

172.1. 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

172.2. Overview

In this portion of the assignment you will be implementing a DTO class that will be used to represent the registration of a racer in a race. All information expressed in the Registration will be derived from the Race and Racer objects — except for the ID.

RacerRegistration.id Avoids Compound Primary Key



The RacerRegistrationDTO `id` was added to keep from having to use a compound (`raceId`, `racerId`) primary key. This makes it an easier 1:1 example with Race and Racer to follow. However, you are free to leverage the `raceId` and `racerId` as a unique primary key here and in the future.

Lecture/Assignment Module Ordering



It is helpful to have a data model in place before writing your various services. However, the lectures are structured with a content-less domain up front and focus 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 classes with at least an `id` field. Skip the details of this section until we have covered the Web content lecture.

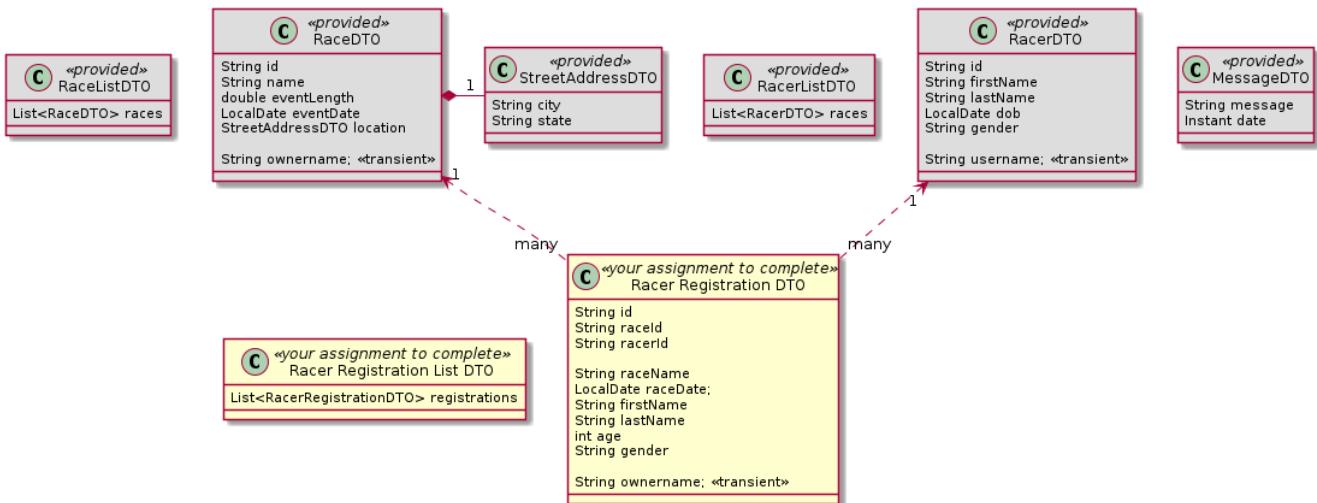


Figure 60. Content

The provided **race-support-api-client** module has the remaining DTO classes.

- **RaceDTO** - provides information specific to the race
- **RacerDTO** - provides information specific to the racer
- **StreetAddress** - provides properties specific to a location
- **MessageDTO** - commonly used to provide error message information for request failures
- <Type>ListDTO - used used to conveniently express typed lists of objects

MessageDTO is from ejava-dto-util Class Examples



The **MessageDTO** class 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 RaceRegistrations portion of the assignment.

172.3. Requirements

1. Create a DTO class to represent **RaceRegistration**
 - a. use the attributes in the diagram above as candidate properties for each class
 - b. **RaceRegistration.age** should be a calculation of years, rounded down, between the **Race.eventDate** and **Racer.dob** properties.
2. Map each DTO class to:
 - a. Jackson JSON (**the only required form**)
 - b. *mapping to Jackson XML is optional*
3. Create a unit test to verify your new DTO type(s) can be marshalled/unmarshalled to/from the targeted serialization type.
4. API TODO: Annotate controller methods to consume and produce supported content type(s) when they are implemented.
5. API TODO: Update clients used in unit tests to explicitly only accept supported content type(s) when they are implemented.

172.4. 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).

172.5. Additional Details

1. 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.
2. 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.
3. Pick a set of hierarchical names for your Java packages. The Java packages can be used to either group common business or architectural areas together. Know that my Race and Racer examples chose to keep common business concepts in the same Java package and break that along architectural boundaries when separated into client and service modules.



I believe that placing the controller, service, and repository classes in the same Java package for classes that primarily operate on the same domain object make the organization much simpler to manage.

4. Use of **lombok** is highly encouraged here and can tremendously reduce the amount of code you write for these classes
5. Java **Period** class can easily calculate age in years between two **LocalDates** (**dob** and **eventDate**).
6. The **race-support-api-client** test cases for Race and Racer demonstrate marshalling and unmarshalling DTO classes within a JUnit test. You should create a similar test of your **RacerRegistrationDTO** 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.

7. The `race-support-api-client` and supplied starter unit tests make use of [@Parameterized JUnit tests](#) that allow a single JUnit test method to be executed N times with variable parameters—*pretty cool feature*. Try it.
8. The `race-support-api-client` module also provides a `RaceDTOFactory` and `RacerDTOFactory` that makes it easy for tests and other demonstration code to quickly assemble example instances. You are encouraged to follow that pattern

Chapter 173. Assignment 2c: Resources

173.1. 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

173.2. Overview

In this portion of the assignment you will be identifying a set of resources to implement a racing registration API. Your results will be documented in a @RestController class. There is nothing to test here until there are implemented DTO and service classes.

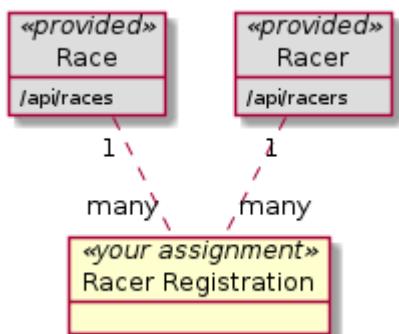


Figure 61. Identify Resources

The API will include three main concepts:

1. **Race** (provided) - an individual race for a specific time and place
 - a. Races can be created, modified, listed, and deleted
 - b. Data for a race can be modified until the first racer registration is added.
 - c. Races can be cancelled or deleted entirely at any time.
2. **Racer** (provided) - identification for a person that may register for races and is not specific to any one race
 - a. Racer information can be created, modified, listed, and deleted
 - b. Racer information can be modified or deleted at any time
3. **Racer Registration** (your assignment) - registration of one racer to a race
 - a. Racer Registrations can be created for an existing race (in the future) and racer
 - b. Any racer information pertinent to race day will be locked into this registration at creation time

- c. Registrations cannot be modified but they can be deleted
- d. A deletion of either the race or racer will eventually cascade to the registration **but not in this assignment**.



Modifications are Replacements

All modifications are replacements. There are no individual field edits requested.

173.3. 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 RacesAPI {  
    public static final String RACES_PATH="/api/races";  
    public static final String RACE_PATH="/api/races/{id}";  
    public static final String RACE_CANCELLATION_PATH=  
        "/api/races/{id}/cancellations";  
    ...  
}
```

- b. create a separate `@RestController` class—at a minimum—for each base resource

Example Skeletal Controller

```
@RestController  
public class RacesController {
```

2. Identify the `@RestController` methods required to represent the following actions. Assign them specific URIs and HTTP methods.
 - a. create new resource
 - b. get a specific resource
 - c. update a specific resource
 - d. list resources with paging
 - e. delete a specific resource
 - f. delete all instances of the resource

Example Skeletal Controller Method

```
@RequestMapping(path=RacesAPI.RACES_PATH,  
    method = RequestMethod.POST,  
    consumes = {...},  
    produces = {...})  
public ResponseEntity<RaceDTO> createRace(@RequestBody RaceDTO newRace) {  
    throw new RuntimeException("not implemented");  
}
```

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

173.4. 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 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

173.5. Additional Details

1. This portion of the assignment alone should produce a set of `@RestController` class(es) with annotated methods that statically define your API interface (possibly missing content details). There is nothing to run or test in this portion.
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

```
public static final String (RESOURCE)S_PATH="(path)";  
public static final 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 — with just an `id` — 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. 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?

Chapter 174. Assignment 2d: Web Client/API Interactions

174.1. 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 Spring MVC RestTemplate or Spring Webflux (in synchronous mode)
3. pass parameters between client and service over HTTP
4. return HTTP response details from service
5. access HTTP response details in client

174.2. Overview

In this portion of the assignment you will invoke your Web API from a client running within a JUnit test case.

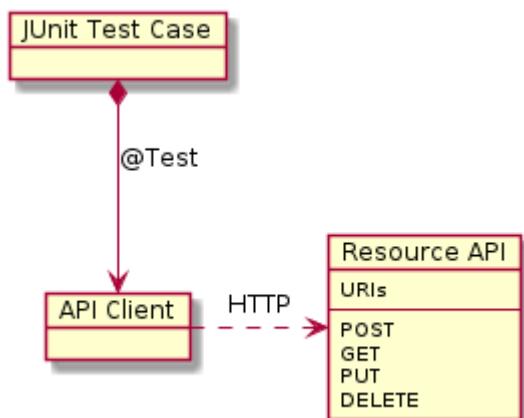


Figure 62. API/Service Interactions

174.3. Requirements

1. Implement stub behavior in the controller class as necessary to complete the example end-to-end calls.

Example Stub Response

```
return ResponseEntity.status(HttpStatus.CREATED)
    .body(RaceDTO.builder()
        .id("1")
        .build());
```

2. Implement a unit integration test to demonstrate a success path
 - a. use either a `RestTemplate` or `WebClient` API client class for this test
 - 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(raceResult.getId()).isNotBlank();
then(raceResult).isEqualTo(raceRequestDTO.withId(raceResult.getId()));
```



Examples use RestTemplate

The Race and Racer examples only use the `RestTemplate` approach.

174.4. 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 Spring MVC `RestTemplate` or Spring `Webflux` (in synchronous mode)
 - a. whether you are able to perform an API call using either the `RestTemplate` or `WebClient` 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

174.5. Additional Details

1. 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 `RestTemplate` or `WebClient` calls. The `race-support-client` module contains example client API classes for Races and Racers using `RestTemplate`.
2. 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.

3. 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.

Chapter 175. Assignment 2e: Service/Controller Interface

175.1. 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

175.2. Overview

In this portion of the assignment you will be implementing the core of the services and integrating them into the controller as seamlessly as possible.

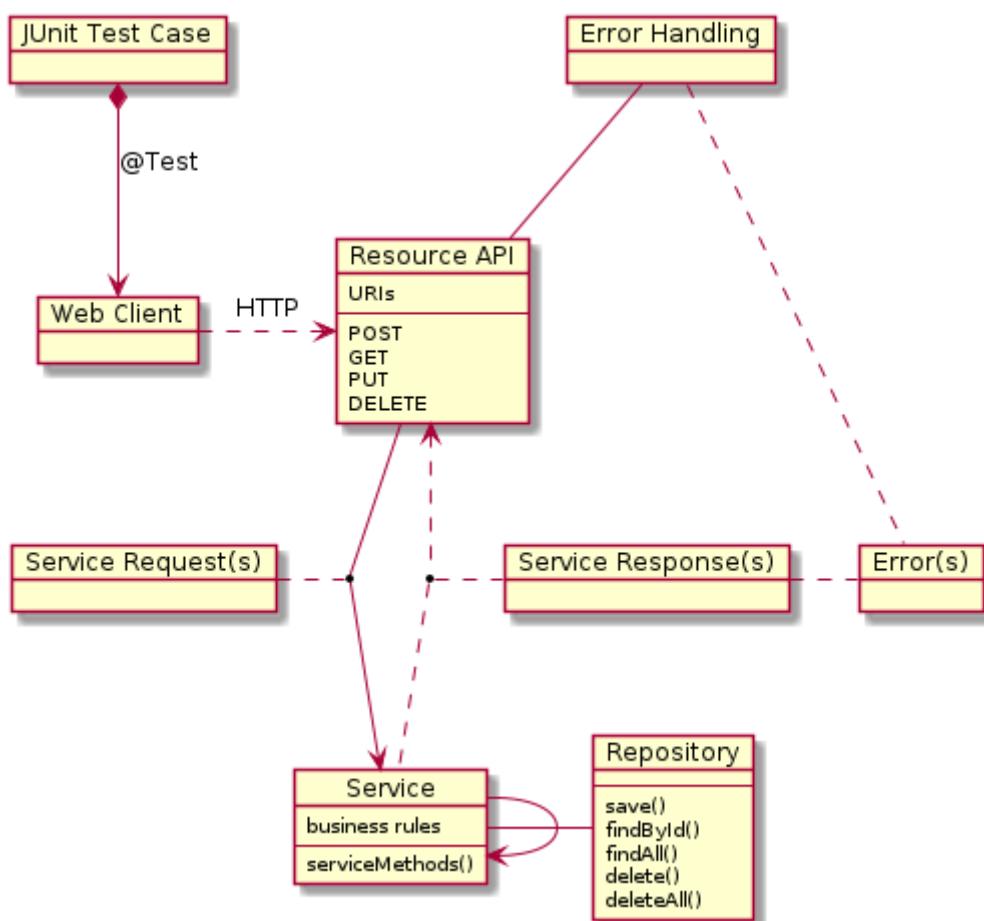


Figure 63. Service/Controller Interface

Your Assignment is Primarily Just RaceRegistration



You have been provided complete implementation of the `Races` and `Racers` services. You only have to implement the `RacerRegistration` components and integration that with `Race` and `Races`.

175.3. Requirements

1. Implement a repository component to simulate necessary behavior (e.g., save, findById) for each base resource type as needed. **Don't go overboard here.** We just need some place to generate IDs and hold the data in memory.
 - a. implement a Java interface for each repository
 - b. implement a component class stub for each repository using simple, in-memory storage (e.g., `HashMap` or `ConcurrentHashMap`) and an ID generation mechanism (e.g., `int` or `AtomicInteger`)
2. Implement a set of services to implement actions and enforce business logic on the base resources
 - a. implement a Java interface for each service
 - b. implement a component class for each service and inject the associated repository
 - c. if there is a service dependency, inject the dependent service(s) as well. For Racer Registration, this includes a dependency on
 - i. `RacesService` - to verify existence of and obtain details of race
 - ii. `RacersService` - to verify existence of and obtain details of racer
 - iii. `Racers Registration Repository` - to store details that are important to the registration
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 issues separate from "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

175.4. 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

175.5. 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. 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 Races and Racers example leverage the exceptions defined in the `ejava-util` module. You are free to define your own.
3. It is highly recommended that `ExceptionHandlers` and `RestExceptionAdvice` be used to handle exceptions thrown and report status. The Races and Racers example leverage the `ExceptionHandlers` from the `ejava-web-util` module. You are free to define your own.

Chapter 176. Assignment 2f: Required Test Scenarios

When all is assembled, there are two minimum scenarios that are required of a complete project.

- successful registration
- failed registration

176.1. Scenario: Successful Registration

In this scenario, a race and racer exist, and the API client is able to successfully create a registration between the two and see the results.



All of the service-side portions of Race and Racer are provided for you. The Race Registration, the assembly, and the test(s) are your primary assignment.

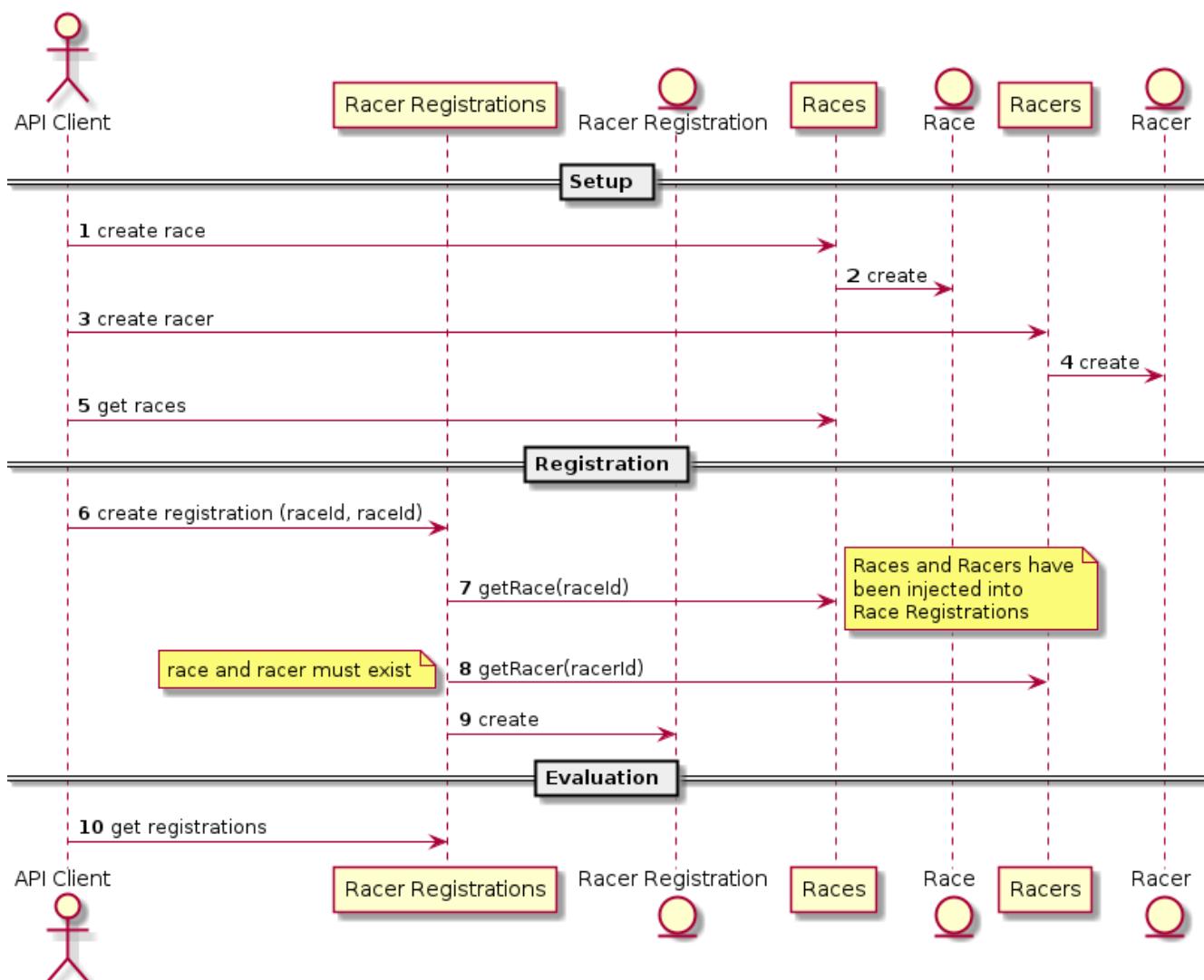


Figure 64. Successful Registration

1. Caller creates race with valid properties

2. (provided) Races generates the raceId and stores the race
3. Caller creates racer with valid properties
4. (provided) Racer generates the racerId and stores the racer
5. The specific race is either known or obtained from Races
6. Register racer for a specific race
7. Race Registration successfully obtains existing race
8. Race Registration successfully obtains existing racer
9. Race Registration creates registration, ID generated, and registration stored
10. Caller obtains race registration information that contains new registration

176.1.1. Alternate Scenario: Race does not Exist

In this alternate scenario, the caller provides the ID of an unknown or deleted/cancelled race. The caller receives a BAD_REQUEST status with a clean error message that is unique to their issue.

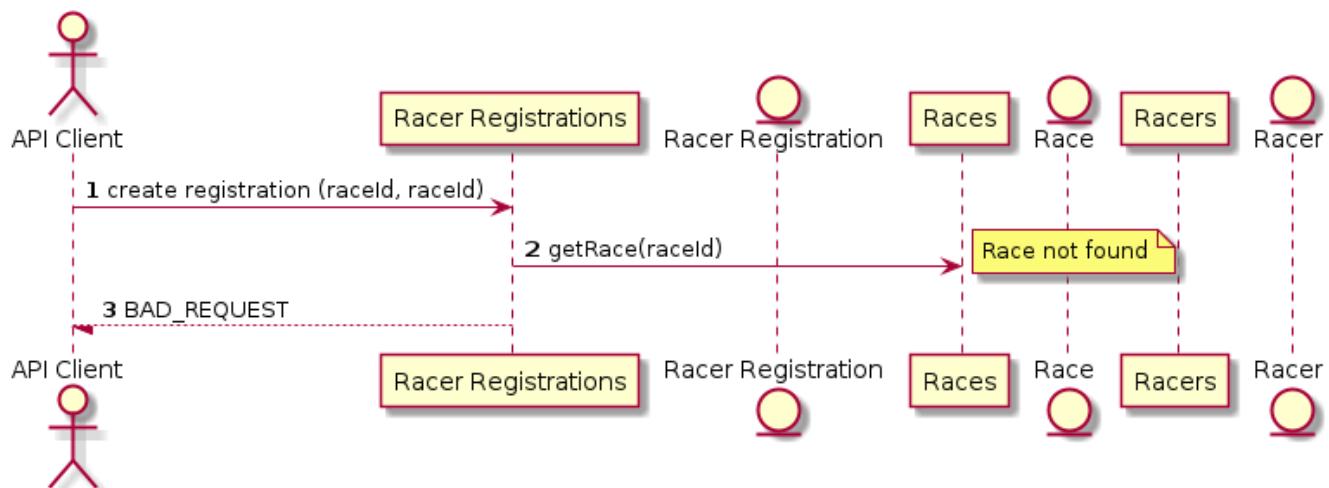


Figure 65. Race does not Exist

1. Caller makes a request to register a racer for a race
2. Race Registration attempts to obtain the race from Races but it does not exist
3. Caller receives an error status of BAD_REQUEST and a **clean** message (not a stack trace) that is unique to the specific request

176.2. Requirements

1. Implement the above 2 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 Race and Racer 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 repository.

176.3. Grading

1. create an integration test that verifies a successful scenario
 - a. whether you implemented an integration unit test that successfully registered a Racer for a Race and produced a RacerRegistration
2. create an integration test that verifies a failure scenario
 - a. whether you implemented an integration unit test that successfully tested an error condition with a RacerRegistration detected on the server-side.

Spring Security Introduction

copyright © 2021 jim stafford (jim.stafford@jhu.edu)

Chapter 177. 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 ^[32] 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.

177.1. 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

177.2. 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. how to 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

[32] "Information Security", Wikipedia

Chapter 178. 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 of the security protections and to be able to develop and leverage mature and potentially certified access control solutions.

Chapter 179. 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.

179.1. Encoding

Encoding converts source information into an alternate form that is safe for communication and/or storage.^[33] 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
amltOnBhc3N3b3Jk
$ echo -n amltOnBhc3N3b3Jk | base64 -D
jim:password
```

① `echo -n` echos the supplied string without new line character added - which would pollute the value

179.2. 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.^[34] Others not in possession of the keys would be forced to try to break the code thru (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
$ openssl enc -aes-256-cbc -salt -in /tmp/plaintext.txt -base64 \①
-pass pass:password > /tmp/ciphertext

$ cat /tmp/ciphertext
U2FsdGVkX18mM2yNc337MS5r/iRJKI+roqkSym0zgMc=

$ openssl enc -d -aes-256-cbc -in /tmp/ciphertext -base64 -pass pass:password ②
jim:password

$ openssl enc -d -aes-256-cbc -in /tmp/ciphertext -base64 -pass pass:password123 ③
bad decrypt
4611337836:error:06FFF064:digital envelope routines:CRYPTO_internal:bad decrypt
```

- ① encrypting file of plaintext with a symmetric/shared key. Result is base64 encoded.
- ② decrypting file of ciphertext with valid symmetric/shared key after being base64 decoded
- ③ failing to decrypt file of ciphertext with invalid key

179.3. 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:$apr1$gLU9VlA1$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.AnEuhUW269ACaPM7tDtD9vbxEg14i9GdGaS

$ htpasswd -bnBC 10 jim password
jim:$2y$10$RztUum5dBjKrcgiBNQ1THueqDFd60RByYgQPbugPCjv23V/RzfdVG

$ htpasswd -bnBC 10 jim password123
jim:$2y$10$s0I8X22Z1k2wK43S7dUBjup2VI1WUaJwfzX8Mg2Ng0jBxnjCEA0F2
```

[33] "Code-Encoding", Wikipedia

[34] "Encryption", Wikipedia

Chapter 180. 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.

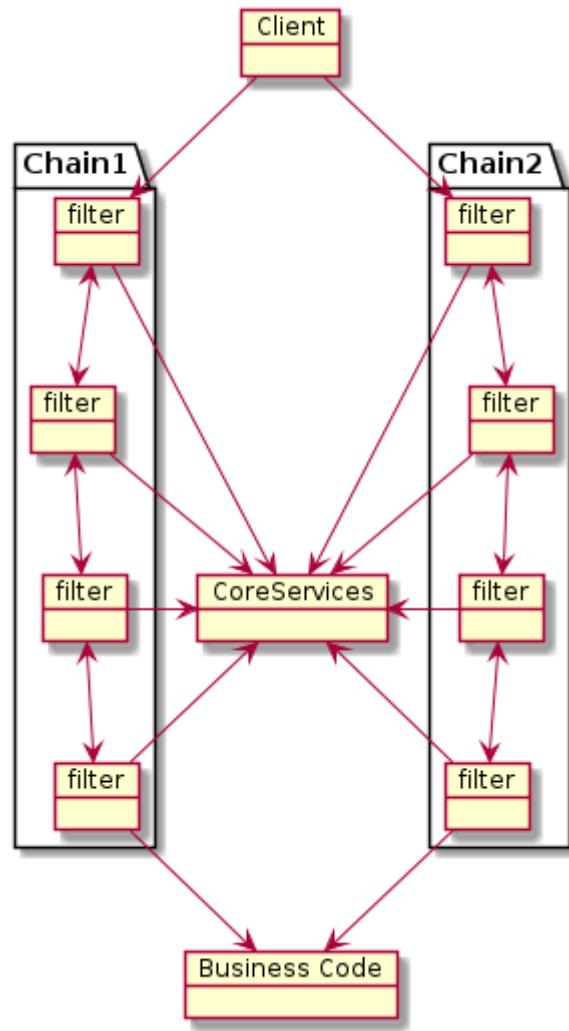


Figure 66. Spring Web Framework Operates thru Flexibly Assembled Filters and Core Services

Chapter 181. 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.

181.1. 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 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
```

181.2. 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.^[35].

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
```

181.3. 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
```

[35] ["Static Content"](#), Spring Boot Reference Documentation

Chapter 182. 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. Lets take a look first at the core services used by the Security Filter Chain.

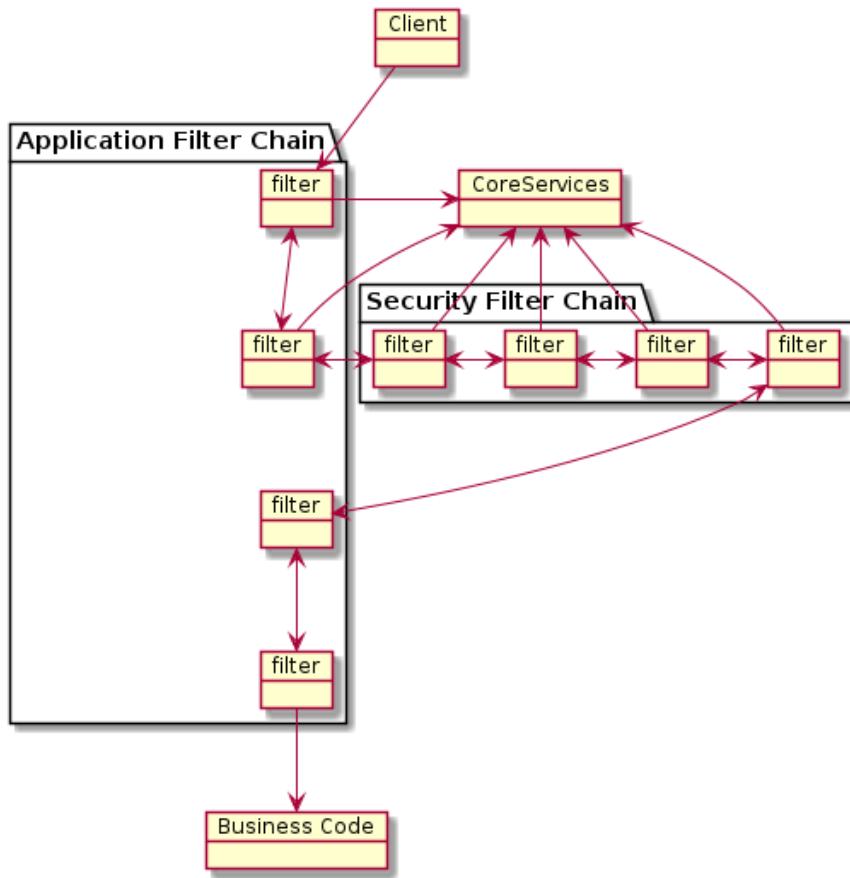


Figure 67. Spring Security Implemented as Extension of Application Filter Chain

182.1. 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.

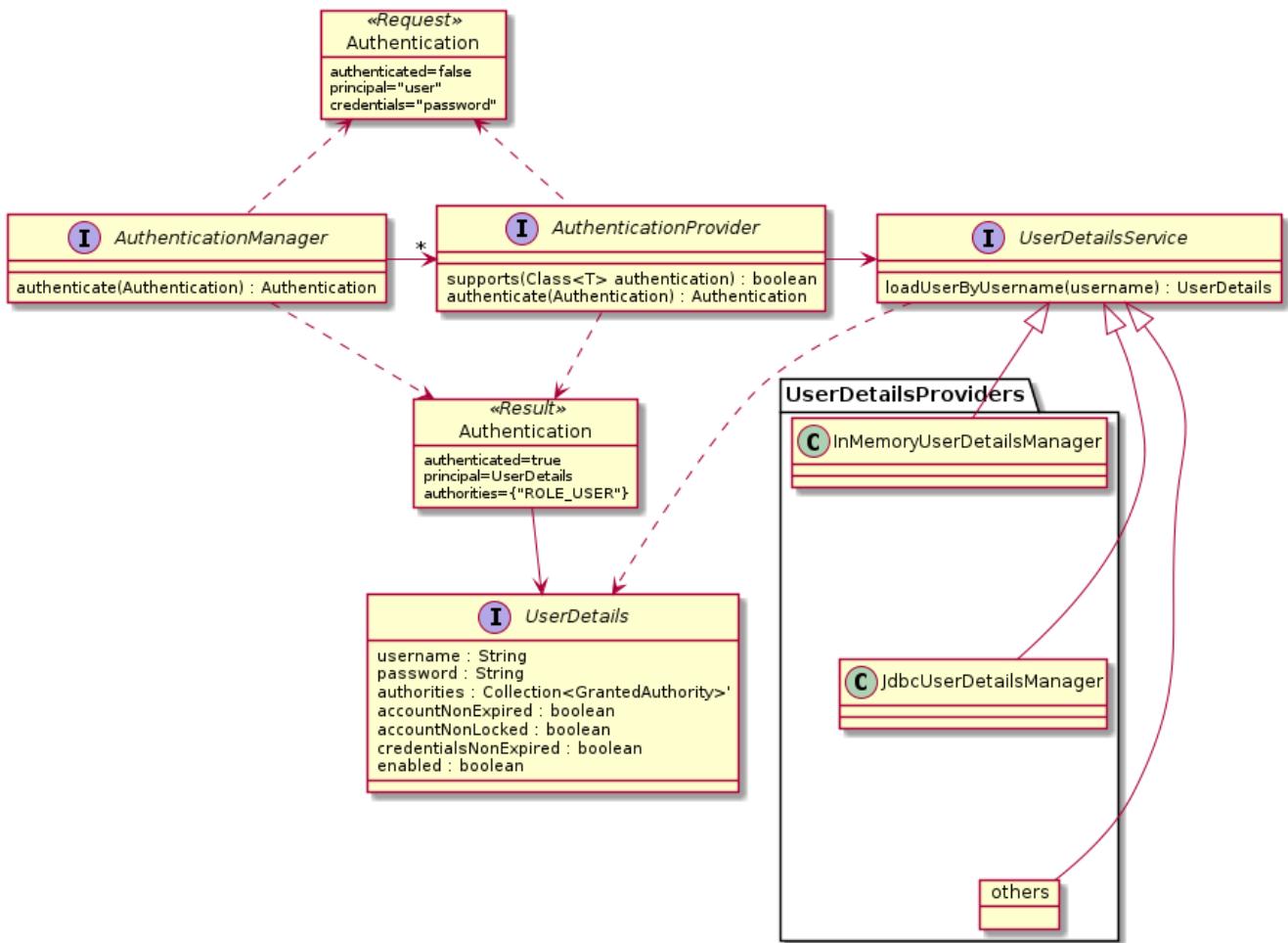


Figure 68. Spring Security Core Authentication Framework

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 the `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.

182.2. SecurityContext

The authentication is maintained inside of 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`.

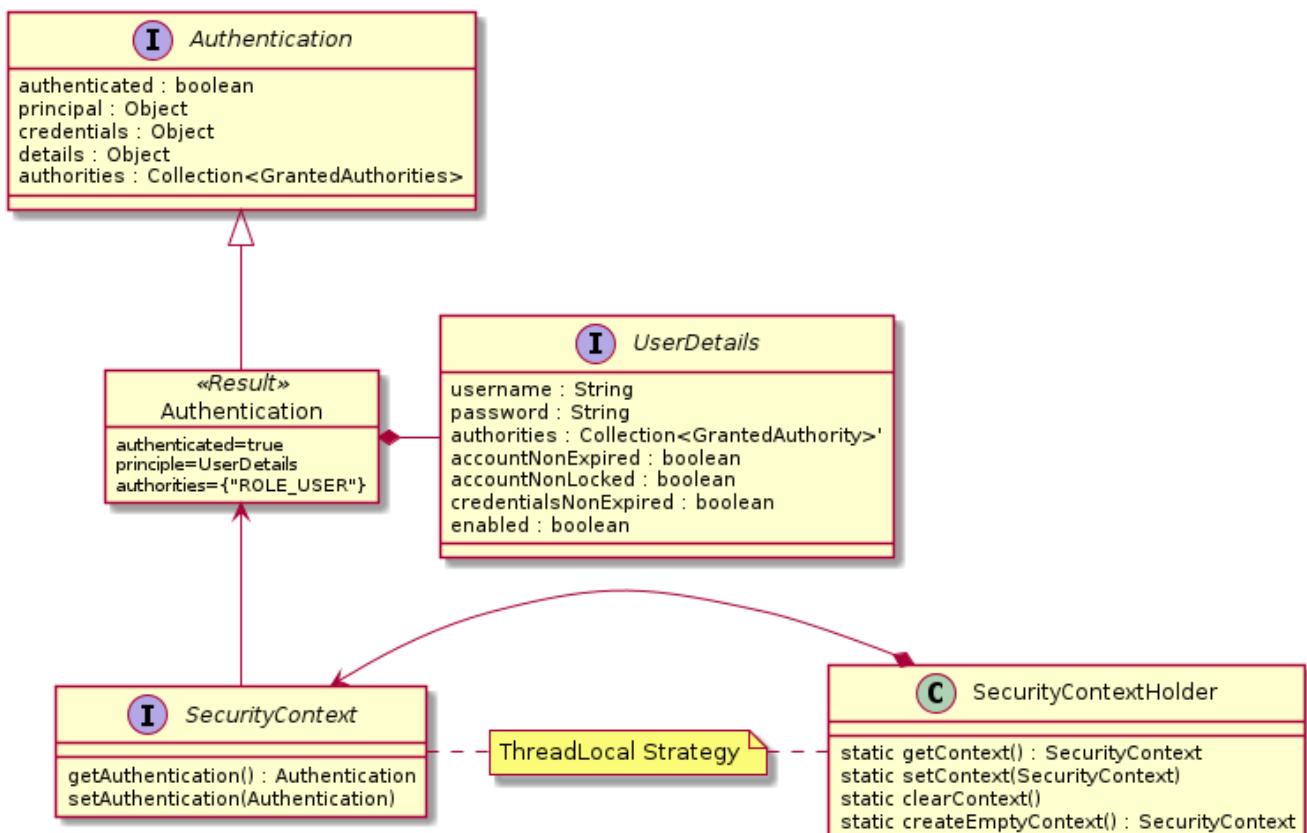


Figure 69. Current `SecurityContext` with `Authentication` accessible through `SecurityContextHolder`

Chapter 183. 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.

183.1. 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.

Spring Boot Starter Security

```
# org.springframework-boot:spring-boot-autoconfigure/META-INF/spring.factories
...
org.springframework.boot.autoconfigure.security.servlet.SecurityAutoConfiguration,\
org.springframework.boot.autoconfigure.security.servlet.UserDetailsServiceAutoConfiguration,\
org.springframework.boot.autoconfigure.security.servlet.SecurityFilterAutoConfiguration,\
```

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.

183.2. 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 `WebSecurityConfigurerAdapter` (more on that later) which
 - requires all URIs be authenticated
 - activates FORM and BASIC authentication
 - enables CSRF and other security protections
- `WebSecurityEnablerConfiguration` which looks to activate all the components by supplying the `@EnableWebSecurity` annotation

183.3. 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: ff40aeeec-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
```

183.4. 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 184. Default FilterChain

When we activated Spring security we added a level of filters that were added 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.

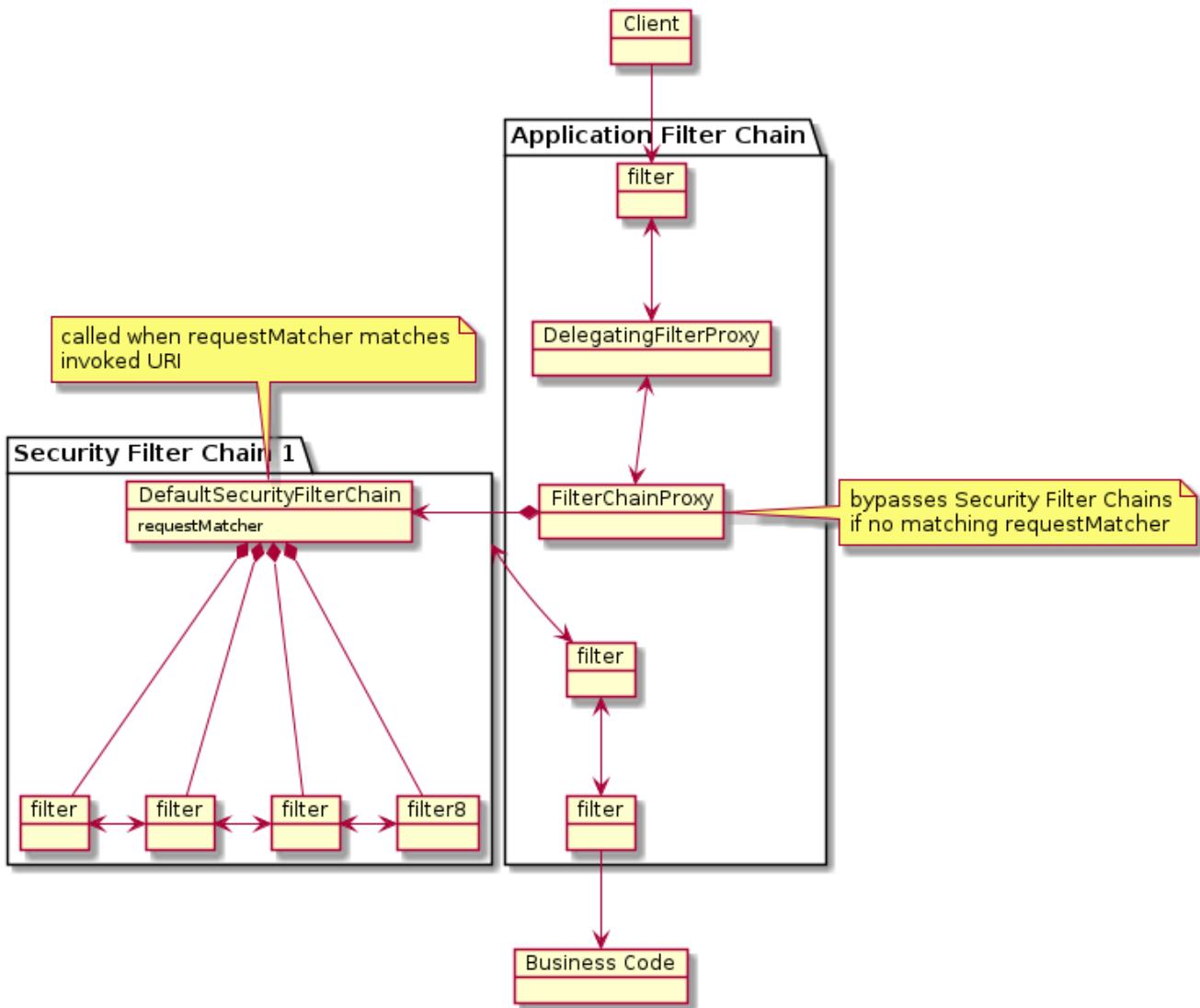


Figure 70. Default Security Filter Chain

Chapter 185. 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 Auto-Configuration left our application.

185.1. 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.

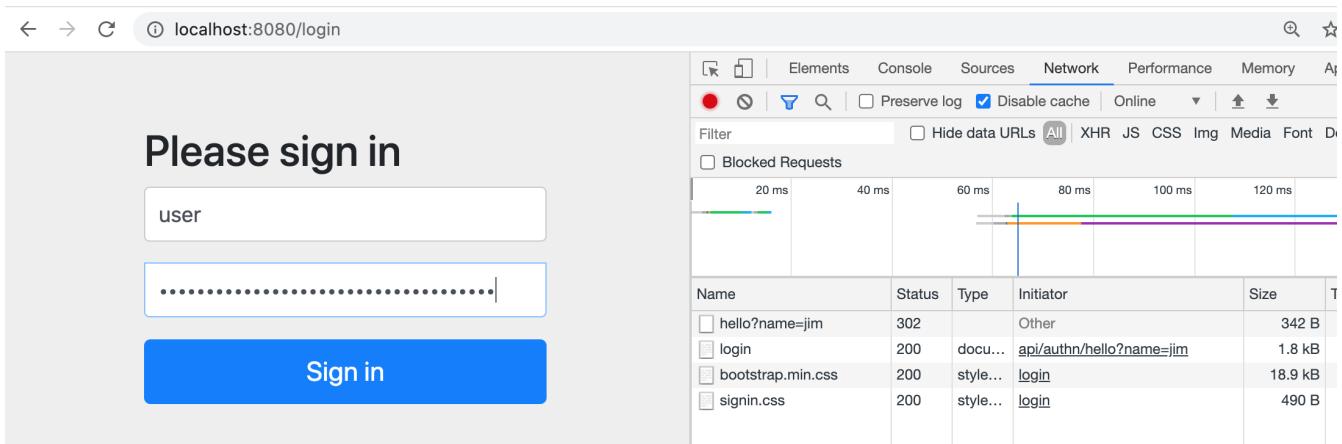


Figure 71. Example Default Form Login Activated

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/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/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.

185.2. 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 a bit more in examples. The following shows an example BASIC encoding of the [username:password](#) values in a Base64 string and then supplying the result of that encoding in an [Authorization](#) header prefixed with the work "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.

185.3. 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  
dXNlcjpiYWRwYXNzd29yZA==  
  
$ curl -v -X GET http://localhost:8080/api/anonymous/hello?name=jim \  
-H "Authorization: BASIC dXNlcjpiYWRwYXNzd29yZA==" ①  
> GET /api/anonymous/hello?name=jim HTTP/1.1  
> Authorization: BASIC dXNlcjpiYWRwYXNzd29yZA==  
>  
< 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

185.4. 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

```
$ echo -n user:password | base64  
dXNlcjpwYXNzd29yZA==  
  
$ curl -v -X GET "http://localhost:8080/api/authn/hello?name=jim" \  
-H "Authorization: BASIC dXNlcjpwYXNzd29yZA=="  
> 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
```

185.5. 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" \
-H "Authorization: BASIC dXNlcjpwYXNzd29yZA==" -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
< Set-Cookie: JSESSIONID=3EEB3625749482AD9E44A3B7E25A0EE4; Path=/; HttpOnly
< WWW-Authenticate: Basic realm="Realm"
< Content-Length: 0
```

185.6. 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 -X GET http://localhost:8080/api/anonymous/hello?name=jim \
-H "Authorization: BASIC dXNlcjpwYXNzd29yZA==" 
> GET /api/anonymous/hello?name=jim HTTP/1.1
> Authorization: BASIC dXNlcjpwYXNzd29yZA==
>
< HTTP/1.1 200
< Set-Cookie: JSESSIONID=EC5EB9D1182F8AC77E290D12AD3BF369; Path=/; HttpOnly
< X-Content-Type-Options: nosniff
< X-XSS-Protection: 1; mode=block
< Cache-Control: no-cache, no-store, max-age=0, must-revalidate
< Pragma: no-cache
< Expires: 0
< X-Frame-Options: DENY
< Content-Type: text/plain; charset=UTF-8
< Content-Length: 10
< Date: Thu, 02 Jul 2020 10:45:32 GMT
<
hello, jim
```

Set-Cookie

a command header to set a small amount of information in the browser to be returned to the server on follow-on calls.^[36] This permits the server to keep track of a user session so that a login state can be retained on follow-on calls.

X-Content-Type-Options

informs the browser that supplied **Content-Type** header responses have been deliberately assigned^[37] and to avoid **Mime Sniffing**—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^[38]

Cache-Control

a header that informs the client how the data may be cached.^[39] 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.^[40]

Expires

a header that contains the date/time when the data should be considered stale and should be re-validated with the server.^[41]

X-Frame-Options

informs the browser whether the contents of the page can be displayed in a frame.^[42] This helps prevent site content from being hijacked in an unauthorized manner. This will not be pertinent to our API responses.

[36] "Using HTTP cookies",MDN web docs

[37] "X-Content-Type-Options", MDN web docs

[38] "X-XSS-Protection Header",OWASP Cheat Sheet Series

[39] "Cache-Control",MDN web docs

[40] "Pragma",MDN web docs

[41] "X-Frame-Options",MDN web docs

[42] "Expires",MDN web docs

Chapter 186. Default FilterChainProxy Bean

The above behavior was put in place by the default Security Auto-Configuration—which is primarily placed within an instance of the `FilterChainProxy` class ^[43]. 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.

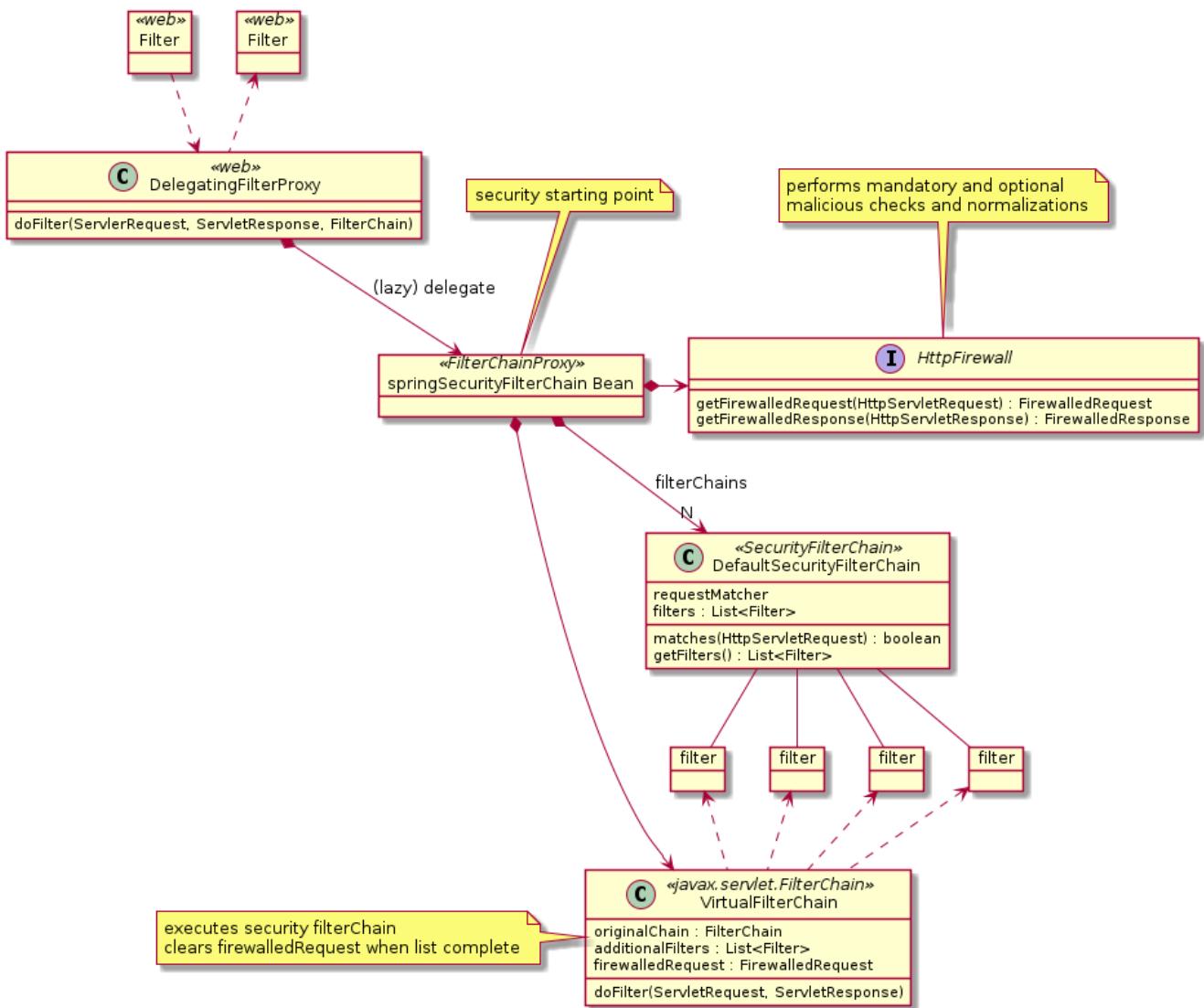


Figure 72. Spring FilterChainProxy Configuration

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/noauth-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>
```

- a breakpoint set on "`FilterChainProxy.doFilterInternal()`" to clearly display the list of filters that will be used for the request.

```

194     private void doFilterInternal(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException, ServletException {
195         FirewalledRequest firewallRequest = this.firewall.getFirewalledRequest((HttpServletRequest) request);
196         HttpServletResponse firewallResponse = this.firewall.getFirewalledResponse((HttpServletResponse) response);
197         List<Filter> filters = getFilters(firewallRequest);
198         filters: size = 15
199         if (filters == null || filters.size() == 0) {
200             if (logger.isTraceEnabled()) {
201                 logger.trace(LogMessage.of(() -> "No security for " + requestLine(firewallRequest)));
202             }
203             firewallRequest.reset();
204             chain.doFilter(firewallRequest, firewallResponse);
205             return;
206         }
207         if (logger.isDebugEnabled()) {
208             logger.debug(LogMessage.of(() -> "Securing " + requestLine(firewallRequest)));
209         }
210         VirtualFilterChain virtualFilterChain = new VirtualFilterChain(firewallRequest, chain, filters);
211     }

```

Variables

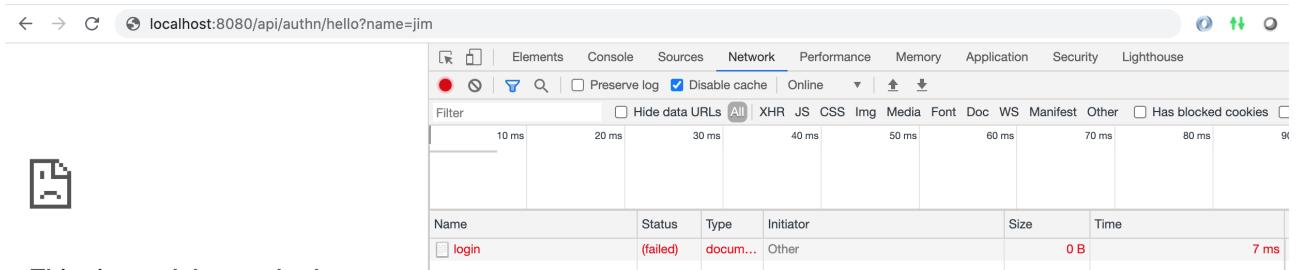
- request = {ServletRequest@6400}
- response = {ResponseFacade@6401}
- chain = {ApplicationFilterChain@6402}
- firewallRequest = {StrictHttpFirewall\$StrictFirewalledRequest@6403} "FirewalledRequest[org.apache.catalina.connector.RequestFacade@7b19f3af]"
- firewallResponse = {FirewalledResponse@6404}
- filters = {ArrayList@6405} size = 15
 - 0 = {WebAsyncManagerIntegrationFilter@7970}
 - 1 = {SecurityContextPersistenceFilter@7971}
 - 2 = {HeaderWriterFilter@7972}
 - 3 = {CsrfFilter@7973}
 - 4 = {LogoutFilter@7974}
 - 5 = {UsernamePasswordAuthenticationFilter@7975}
 - 6 = {DefaultLoginPageGeneratingFilter@7976}
 - 7 = {DefaultLogoutPageGeneratingFilter@7977}
 - 8 = {BasicAuthenticationFilter@7978}
 - 9 = {RequestCacheAwareFilter@7979}
 - 10 = {SecurityContextHolderAwareRequestFilter@7980}
 - 11 = {AnonymousAuthenticationFilter@7981}
 - 12 = {SessionManagementFilter@7982}
 - 13 = {ExceptionTranslationFilter@7983}
 - 14 = {FilterSecurityInterceptor@7984}

- another breakpoint set on "`FilterChainProxy.VirtualFilterChain.doFilter()`" to pause in between each filter.

```

319
320     * 
321     * @Override
322     * public void doFilter(ServletRequest request, ServletResponse response) throws IOException, ServletException {
323         if (this.currentPosition == this.size) {
324             if (logger.isDebugEnabled()) {
325                 logger.debug(LogMessage.of(() -> "Secured " + requestLine(this.firewalledRequest)));
326             }
327             // Deactivate path stripping as we exit the security filter chain
328             this.firewalledRequest.reset(); firewalledRequest: "FirewalledRequest[ org.apache.catalina.connector.Request@6402
329             this.originalChain.doFilter(request, response); originalChain: ApplicationFilterChain@6402
330             return;
331         }
332         this.currentPosition++;
333         Filter nextFilter = this.additionalFilters.get(this.currentPosition - 1); nextFilter: WebAsyncManagerIntegrationFilter@7970
334         if (logger.isTraceEnabled()) {
335             logger.trace(LogMessage.of(format: "Invoking %s (%d/%d)", nextFilter.getClass().getSimpleName(),
336                                         this.currentPosition, this.size)); currentPosition: 1 size: 15
337         }
338     }
339 }
```

- 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.

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.

SecurityContextPersistenceFilter

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.

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)

LogoutFilter

Looks for calls to logout 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.

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

Assigns anonymous use to security context if no user is identified

SessionManagementFilter

Performs any required initialization and security checks in order to setup the current session

ExceptionTranslationFilter

Attempts to augment any thrown `AccessDeniedException` and `AuthenticationException` with details related to the denial. 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.

FilterSecurityInterceptor

Applies the authenticated user against access constraints. It throws an `AccessDeniedException` if

denied, which is caught by the `ExceptionTranslationFilter`.

This is also where the security filter chain hands control over to the application filter chain where the endpoint will get invoked.

[43] "`FilterChainProxy`", Spring Security Reference Manual

Chapter 187. 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 © 2021 jim stafford (jim.stafford@jhu.edu)

Chapter 188. 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.

188.1. 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

188.2. 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 client
8. add authentication to integration tests

Chapter 189. WebSecurityConfigurer

To eliminate the defaults and define a customized `FilterChainProxy`-- we must supply one or more classes that implement the `WebSecurityConfigurer` interface. That is made easier by extending the `WebSecurityConfigurerAdapter` class— which is the class that supplies all the defaults we leveraged in the last chapter.

To highlight that the `FilterChainProxy` is populated with a prioritized list of `SecurityFilterChain`—I am going to purposely create multiple `WebSecurityConfigurer` beans.

- 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 well we can customize Spring Security.

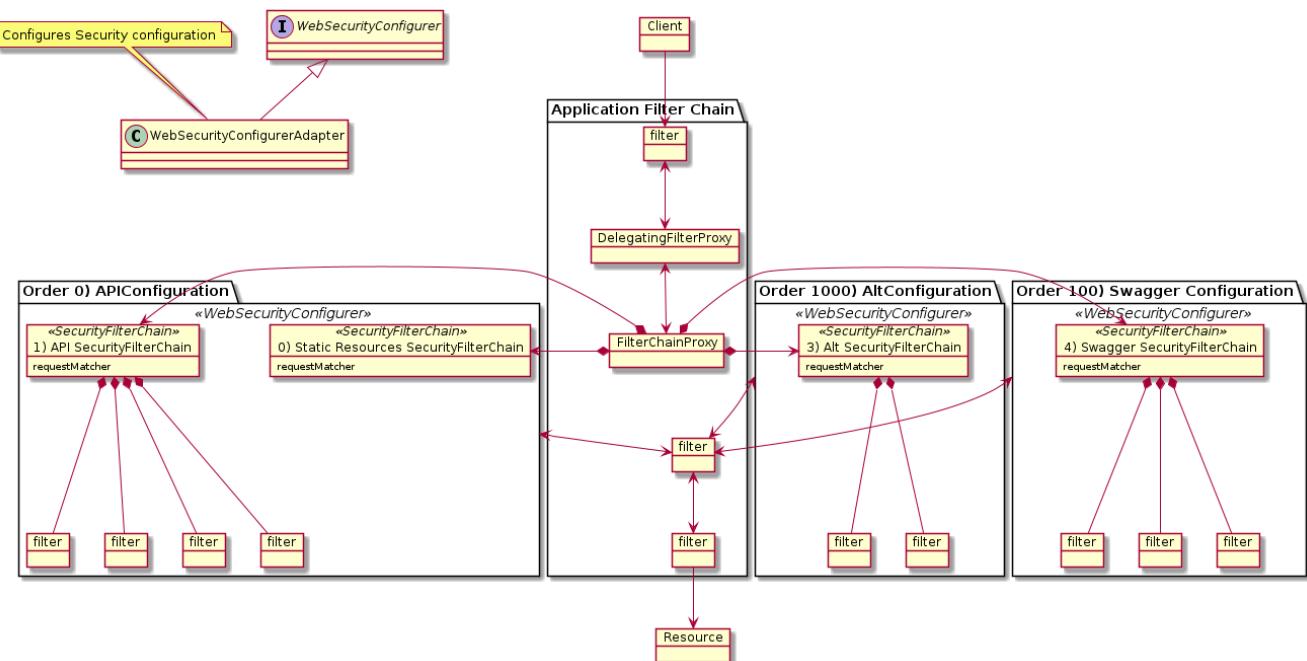


Figure 73. Multiple SecurityFilterChains

189.1. Core Application Security Configuration

I have purposely created an outer and inner set of `@Configuration` classes for the core application security to help demonstrate a point that there can be multiple `SecurityFilterChain` and multiple distinct configurations for them. Take a look at the example below.

- the outer `@Configuration` class will contain `@Bean` definitions for anything deemed globally shared (e.g., passwordEncoder, user storage)
- the inner `@Configuration` classes will define purpose-specific `SecurityFilterChain` to ultimately be part of the `FilterChainProxy`

The priority of each SecurityFilterChain is determined by the `@Order` annotation placed on the `@Configuration` class. Note that the later `@Configuration` is 100% in the snippet and is provided to show default behavior against the `/api/alt` URI after we have customized the other two.

Core Application Security Configuration

```
package info.ejava.examples.svc.authn.authcfg.security;

import org.springframework.context.annotation.Configuration;
import org.springframework.core.annotation.Order;
import
org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurer
Adapter;
...

@Configuration
public class SecurityConfiguration {

    @Configuration
    @Order(0) ②
    public static class APIConfiguration extends WebSecurityConfigurerAdapter { ①
        //...
    }

    @Configuration
    @Order(1000) ②
    public static class AltConfiguration extends WebSecurityConfigurerAdapter { ③
    }
}
```

- ① Create `@Configuration` class that extends `WebSecurityConfigurerAdapter` to customize `SecurityFilterChain`
- ② `APIConfiguration` has a higher priority and higher priority resulting `SecurityFilterChain`s than `AltConfiguration`
- ③ Default behavior re-instantiated and applied to anything not handled by higher priority configurations

WebSecurityConfigurerAdapter can create multiple FilterChainProxy



Depending on what is supplied—a single `WebSecurityConfigurerAdapter` may result in multiple `FilterChainProxy`.

189.2. Additional Swagger Security Configuration

Once we enabled default security on our application—we lost the ability to access the Swagger page without logging in. We did not have to create a separate `WebSecurityConfigurerAdapter` for just the Swagger endpoints—but doing so provides some nice modularity and excuse to further demonstrate Spring security configurability.

Swagger Security Configuration

```
package info.ejava.examples.svc.authn;

import org.springframework.boot.autoconfigure.condition.ConditionalOnClass;
import org.springframework.boot.autoconfigure.condition.ConditionalOnProperty;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.annotation.Order;
import
org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurer
Adapter;

@Configuration
@ConditionalOnProperty(name = "test", havingValue = "false", matchIfMissing = true) ①
public class SwaggerConfiguration {
    //...
    @Configuration
    @Order(100) ③
    @ConditionalOnClass(WebSecurityConfigurerAdapter.class) ②
    public static class SwaggerSecurity extends WebSecurityConfigurerAdapter {
        //...
    }
}
```

① Configuration can be turned off with the presence of `test=true` property

② Configuration will be ignored at runtime if `WebSecurityConfigurerAdapter` class does not exist (i.e., security was not added)

③ Priority (100) is after core application (0) and prior to default rules (1000)

189.3. 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.

Ignore Static Content Configuration

```
import org.springframework.security.config.annotation.web.builders.WebSecurity;

@Configuration
@Order(0)
public static class APIConfiguration extends WebSecurityConfigurerAdapter {
    @Override
    public void configure(WebSecurity web) throws Exception {
        web.ignoring().antMatchers("/content/**");
    }
}
```

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.

Anonymous Access to Static Content

```
$ 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
```

189.4. SecurityFilterChain Matcher

The meat of the configuration is within the `configure(HttpSecurity)` method. It is here where we impact the well-populated `SecurityFilterChain`. The configuration is determined by calls to the provided `HttpSecurity` object. In the example below I am limiting the configuration to two URIs (`/api/anonymous` and `/api/authn`) using an Ant Matcher. A regular expression matcher is also available. The matchers also allow a specific method to be declared in the definition.

SecurityFilterChain Matcher

```
import org.springframework.security.config.annotation.web.builders.HttpSecurity;

@Configuration
@Order(0)
public static class APIConfiguration extends WebSecurityConfigurerAdapter {
    ...
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.requestMatchers(m->m.antMatchers("/api/anonymous/**", "/api/authn/**"));①
        // ...
    }
}
```

① rules within this configuration will apply to URIs below `/api/anonymous` and `/api/authn`

189.5. HttpSecurity Builder Methods

The `HttpSecurity` object is "builder-based" and has several options on how it can be called.

- `http.methodReturningBuilder().configureBuilder()`
- `http.methodPassingBuilder(builder->builder.configureBuilder())`

The builders are also designed to be chained. It is quite common to see the following syntax used.

Chained Builder Calls

```
http.authorizeRequests()
    .anyRequest()
    .authenticated()
    .and().formLogin()
    .and().httpBasic();
```

We can simply make separate calls. 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. You will see me using separate calls with the pass-the-builder and configure with a lambda style. Either style functionally works the same.

Separate Builder Calls with Lambdas

```
http.authorizeRequests(cfg->cfg.anyRequest().authenticated());
http.formLogin();
http.httpBasic();
```

189.6. 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

```
@Override  
protected void configure(HttpSecurity http) throws Exception {  
    ...  
    http.authorizeRequests(cfg->cfg.antMatchers("/api/anonymous/**").permitAll());  
    http.authorizeRequests(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

189.7. 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`.

```
@Override  
protected void configure(HttpSecurity http) throws Exception {  
    ...  
    http.httpBasic(cfg->cfg.realmName("AuthConfigExample"));  
    http.formLogin(cfg->cfg.disable());
```

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

189.8. 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 [44] 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

```
@Override  
protected void configure(HttpSecurity http) throws Exception {  
    ...  
    http.headers(cfg->{  
        cfg.xssProtection().disable();  
        cfg.frameOptions().disable();  
    });  
    http.csrf(cfg->cfg.disable());  
}
```

189.9. 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

```
@Override  
protected void configure(HttpSecurity http) throws Exception {  
    ...  
    http.sessionManagement(cfg->  
        cfg.sessionCreationPolicy(SessionCreationPolicy.STATELESS));  
}
```

[44] "X-XSS-Protection",MDN Web Docs

Chapter 190. 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.

190.1. 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

190.2. Successful Authenticated Call

The following shows a successful authenticated call and the returned headers.

Successful Authenticated Call

```
$ echo -n user:password | base64 ①
dXNlcjpwYXNzd29yZA==

$ curl -v -X GET http://localhost:8080/api/authn/hello?name=jim \
-H "Authorization: BASIC dXNlcjpwYXNzd29yZA=="
> 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

190.3. 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 191. Authenticated User

Authenticating the identity of the caller is a big win. We likely will want their identity at some point during the call.

191.1. 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.

```
import org.springframework.security.core.annotation.AuthenticationPrincipal;
import org.springframework.security.core.userdetails.UserDetails;
...
public String getHello(@RequestParam String name,
                      @AuthenticationPrincipal UserDetails user) {
    return "hello, " + name + " :caller=" + (user==null ? "(null)" : user.getUsername());
}
```

191.2. Obtain SecurityContext from Holder

The other option is to lookup 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.

```
import org.springframework.security.core.context.SecurityContextHolder;

public String getHelloAlt(@RequestParam String name) {
    UserDetails user = (UserDetails) SecurityContextHolder
        .getContext().getAuthentication().getPrincipal();
    return "hello, " + name + " :caller=" + user.getUsername();
}
```

Chapter 192. Swagger BASIC Auth Configuration

I have added a separate security configuration for the OpenAPI and Swagger endpoints.

192.1. Swagger Authentication Configuration

The following configuration allows the OpenAPI and Swagger endpoints to be accessed anonymously and handle authentication within OpenAPI/Swagger.

Swagger

```
@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());
    }
}
```

192.2. 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 getHelloAuthn(@RequestParam 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, it displays "OpenAPI definition v0 OAS3" and a "Servers" dropdown set to "http://localhost:8080 - Generated server url". A red circle highlights the "Authorize" button in the top right corner. Below this, under the heading "hello-controller", it says "demonstrates sample calls with security constraints". A dropdown menu is open next to the heading. The main content area lists several API endpoints:

- GET /api/anonymous/hello**
- POST /api/anonymous/hello**
- GET /api/authn/hello** (with a lock icon)
- POST /api/authn/hello** (with a lock icon)
- GET /api/alt/hello** (with a lock icon)
- POST /api/alt/hello** (with a lock icon)

Figure 74. 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 'Parameters' section contains a single parameter named 'name' (type: string, query) with the value 'jim'. Below the parameters are two buttons: 'Execute' (blue) and 'Clear' (grey). The 'Responses' section is collapsed. The 'Curl' section displays a command-line example:

```
curl -X GET "http://localhost:8080/api/authn/hello?name=jim" -H "accept: */*" -H "Authorization: Basic dXNlcjpwYXNzd29yZA=="
```

A red oval highlights the 'Authorization' header line in the curl command.

Figure 75. Swagger BASIC Auth Call

Chapter 193. 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). 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.

193.1. Origin Request Header

Implementing CORS checks, browsers will add an additional **Origin** header with the identity of where the code was loaded from if that does not match the base URL (**scheme://host:port**) of where the loaded code is calling.

CORS Response Allowing Calls from Origin

```
$ curl -v http://localhost:8080/api/anonymous/hello?name=jim \②
-H "Origin: http://acme.com"
> GET /api/anonymous/hello?name=jim HTTP/1.1
> Origin: http://acme.com ① ③
>
< HTTP/1.1 200
< Vary: Origin
< Vary: Access-Control-Request-Method
< Vary: Access-Control-Request-Headers
< Access-Control-Allow-Origin: http://acme.com ④
hello, jim :caller=(null)
```

① code making a call from the browser was loaded from <http://acme.com>

② code is calling different base URL <http://localhost:8080>

③ browser supplies **Origin** header with source of code

④ server configured to accept calls from code loaded from <http://acme.com> and returns an **Access-Control-Allow-Origin** header with either the name of that Origin or wildcard character (*) indicating the call is allowed

193.2. Access-Control-Allow-Origin Response Header

If an **Access-Control-Allow-Origin** header is not in the response it will be blocked from the calling code — whether it be because:

- CORS is not implemented by the server

CORS Not Implemented Response

```
$ curl -v http://localhost:8080/api/anonymous/hello?name=jim \
-H "Origin: http://acme.com"
> GET /api/anonymous/hello?name=jim HTTP/1.1
> Origin: http://acme.com
>
< HTTP/1.1 200
hello, jim :caller=(null)
```

- server actively rejects the request from the Origin

CORS Response Rejecting Calls from Origin

```
$ curl -v http://localhost:8080/api/anonymous/hello?name=jim \
-H "Origin: http://foo.com"
> GET /api/anonymous/hello?name=jim HTTP/1.1
> Origin: http://foo.com
>
< HTTP/1.1 403
< Vary: Origin
< Vary: Access-Control-Request-Method
< Vary: Access-Control-Request-Headers
...
Invalid CORS request
```

193.3. Browser Blocks Response

The browser will not allow the code that invoked the API to receive the response without the **Access-Control-Allow-Origin** present and indicating the call is allowed.



Figure 76. Browser Blocks Response

In the above image, the HTML file was loaded from disk and loaded a javascript file also from disk that made a call to our API.

193.4. Browser Supplied Origin of Javascript

The browser supplied the **Origin** header (value="null"), which did not match the "schema://host:port" of the HTTP request or any registered origin within the application.

Name	Headers	Preview	Response	Initiator	Timing
index.html					
hello.js					
jquery.min.js					
hello?name=jim					
	▼ Response Headers view source Cache-Control: no-cache, no-store, max-age=0, must-revalidate Connection: keep-alive Content-Length: 25 Content-Type: text/plain; charset=UTF-8 Date: Sat, 04 Jul 2020 11:40:55 GMT Expires: 0 Keep-Alive: timeout=60 Pragma: no-cache X-Content-Type-Options: nosniff				
	▼ Request Headers view source Accept: */* Accept-Encoding: gzip, deflate, br Accept-Language: en-US,en;q=0.9 Cache-Control: no-cache Connection: keep-alive Host: localhost:8080 Origin: null Pragma: no-cache Sec-Fetch-Dest: empty Sec-Fetch-Mode: cors Sec-Fetch-Site: cross-site User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_5) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/83.0.4103.116 Safari/537.36				

Figure 77. Browser Supplies Origin of Javascript

193.5. Spring MVC @CrossOrigin Annotation

Spring offers many ways to enable the CORS protocol. The first option shown is a `@CrossOrigin` annotation added to the controller class or individual operations indicating CORS constraints.

This technique is static and likely only used to open the API up to all origins.

Spring MVC @CrossOrigin Annotation

```
...
import org.springframework.web.bind.annotation.CrossOrigin;
...
@CrossOrigin ①
@RestController
public class HelloController {
```

① defaults to all origins, etc.

193.6. Spring Security CORS Filter

Spring Security filters also offer a dynamic way to implement the CORS protocol—with the chance to inspect each call. We can register the filter in one of two ways

- through the application context
- directly using the builder

```
protected void configure(HttpSecurity http) throws Exception {  
    //...  
    http.cors(); ①  
    //http.cors(cfg -> cfg.configurationSource(corsConfigurationSource()));②  
}  
...  
  
@Bean  
public CorsConfigurationSource corsConfigurationSource() {  
    return new CorsConfigurationSource() {  
        @Override  
        public CorsConfiguration getCorsConfiguration(HttpServletRequest request) {  
            ③  
        }  
    };  
    ...  
}
```

① enables CORS filter and looks for a `CorsConfigurationSource` bean with the name `corsConfigurationSource`

② enables CORS filter and directly registers the configuration with the builder

③ CORS configuration is determined within the scope of an individual request

193.7. Dynamically Permit All CORS Requests

We can repeat the functionality of the `@CrossOrigin` annotation by simply returning a `CorsConfiguration` instance with `applyPermitDefaultValues()`. This type of configuration would likely be in place for development.

Permit All CORS Requests Configuration

```
@Bean  
public CorsConfigurationSource corsConfigurationSource() {  
    return new CorsConfigurationSource() {  
        @Override  
        public CorsConfiguration getCorsConfiguration(HttpServletRequest request) {  
            CorsConfiguration config = new CorsConfiguration();  
            config.applyPermitDefaultValues(); ①  
            return config;  
        }  
    };  
}
```

① all requests will be accepted

193.8. Custom CORS Configuration

We can also return a custom configuration. This type of configuration would likely be in place for production—except with a more thorough examination of the Origin header.

Custom CORS Configuration

```
@Bean
public CorsConfigurationSource corsConfigurationSource() {
    return new CorsConfigurationSource() {
        @Override
        public CorsConfiguration getCorsConfiguration(HttpServletRequest request) {
            CorsConfiguration config = new CorsConfiguration();
            config.addAllowedOrigin("http://acme.com"); ①
            config.setAllowedMethods(Arrays.asList("GET", "POST"));
            return config;
        }
    };
}
```

① more refined definition of CORS configuration on per-call basis



@Bean instances can be profile-specific

By placing the CorsConfigurationSource in a `@Bean`, we can make that bean configurable based on profiles or other choices. That means we can have a lenient configuration for development that is independent of integration and production environments.

193.9. Successful Browser/Server CORS Exchange

The following shows a successful browser/server CORS exchange — where the server has been configured to allow calls from any Origin and the browser reported a "null" Origin when loading the javascript from disk. The response is given the javascript and shown in the browser.

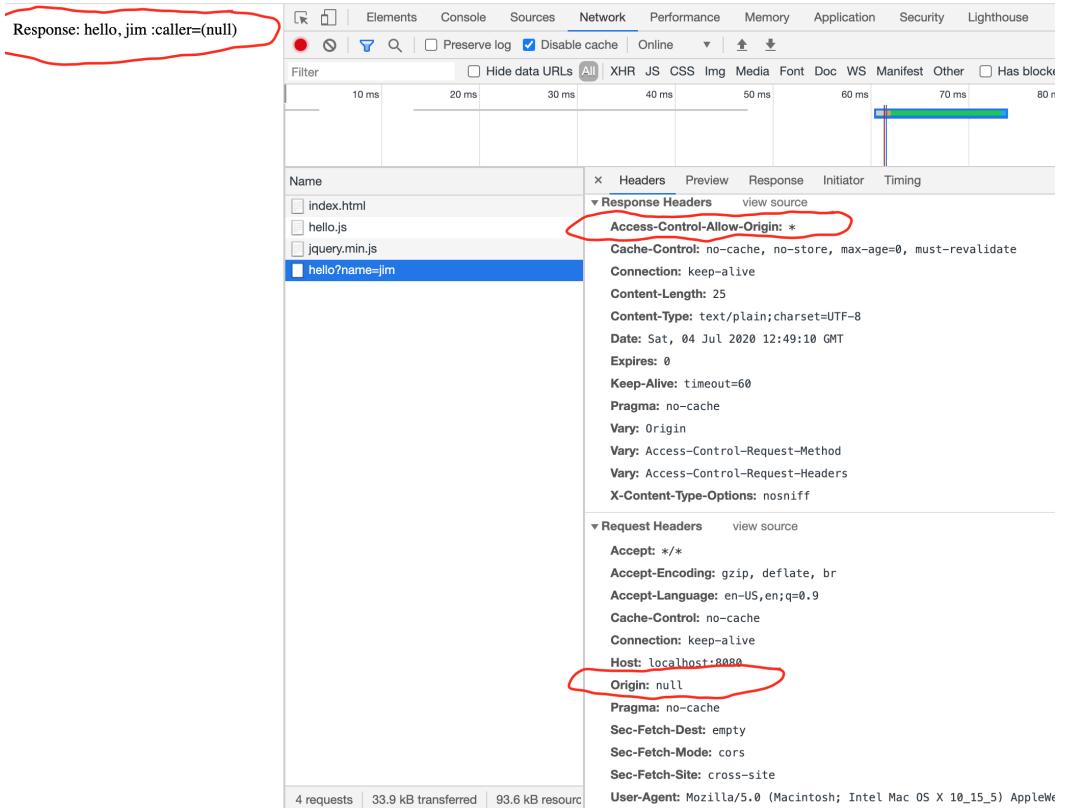


Figure 78. Successful Browser/Server CORS Exchange

Chapter 194. 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.

RestTemplate Anonymous Client

```
@Bean
public RestTemplate anonymousUser(RestTemplateBuilder builder) {
    RestTemplate restTemplate = builder.requestFactory(
        //used to read the streams twice -- so we can use the logging filter below
        ()->new BufferingClientHttpRequestFactory(
            new SimpleClientHttpRequestFactory()))
        .interceptors(new RestTemplateLoggingFilter())
        .build(); ①
    return restTemplate;
}
```

① vanilla RestTemplate with our debug log interceptor

RestTemplate Authenticating Client

```
@Bean
public RestTemplate authnUser(RestTemplateBuilder builder) {
    RestTemplate restTemplate = builder.requestFactory(
        //used to read the streams twice -- so we can use the logging filter below
        ()->new BufferingClientHttpRequestFactory(
            new SimpleClientHttpRequestFactory()))
        .interceptors(
            new BasicAuthenticationInterceptor("user", "password"), ①
            new RestTemplateLoggingFilter())
        .build();
    return restTemplate;
}
```

① added BASIC Auth filter to add Authorization Header

194.1. 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={AuthConfigExampleApp.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 195. Mock MVC Authentication

There are many test frameworks within Spring and Spring Boot that I did not cover them all 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 too 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(classes={AuthConfigExampleApp.class},  
    properties = "test=true")  
@AutoConfigureMockMvc  
public class AuthConfigMockMvcNTest {  
    @Autowired  
    private WebApplicationContext context;  
    @Autowired  
    private MockMvc anonymous;  
    //example manual instantiation ①  
    private MockMvc user;  
    private final String uri = "/api/anonymous/hello";  
  
    @BeforeEach  
    public void init() {  
        user = MockMvcBuilders  
            .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

195.1. MockMvc Anonymous Call

The first test is a baseline example showing a call thru the mock to a service that allows all callers and no required authentication.

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)"));  
}
```

195.2. MockMvc Authenticated Call

The next example shows how we can inject an identity into the mock for use during the test method.

MockMvc Authenticated Call

```
@WithMockUser("user")  
@Test  
public void user_can_call_get() throws Exception {  
    user.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.

Chapter 196. 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

User Details

copyright © 2021 jim stafford (jim.stafford@jhu.edu)

Chapter 197. 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.

197.1. 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

197.2. 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 198. 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.

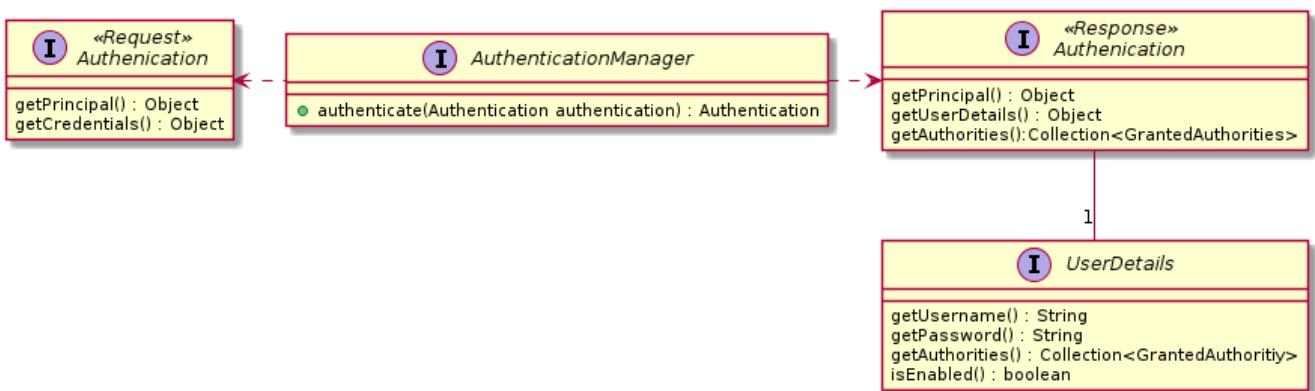


Figure 79. `AuthenticationManager` and `UserDetails`

198.1. 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`.

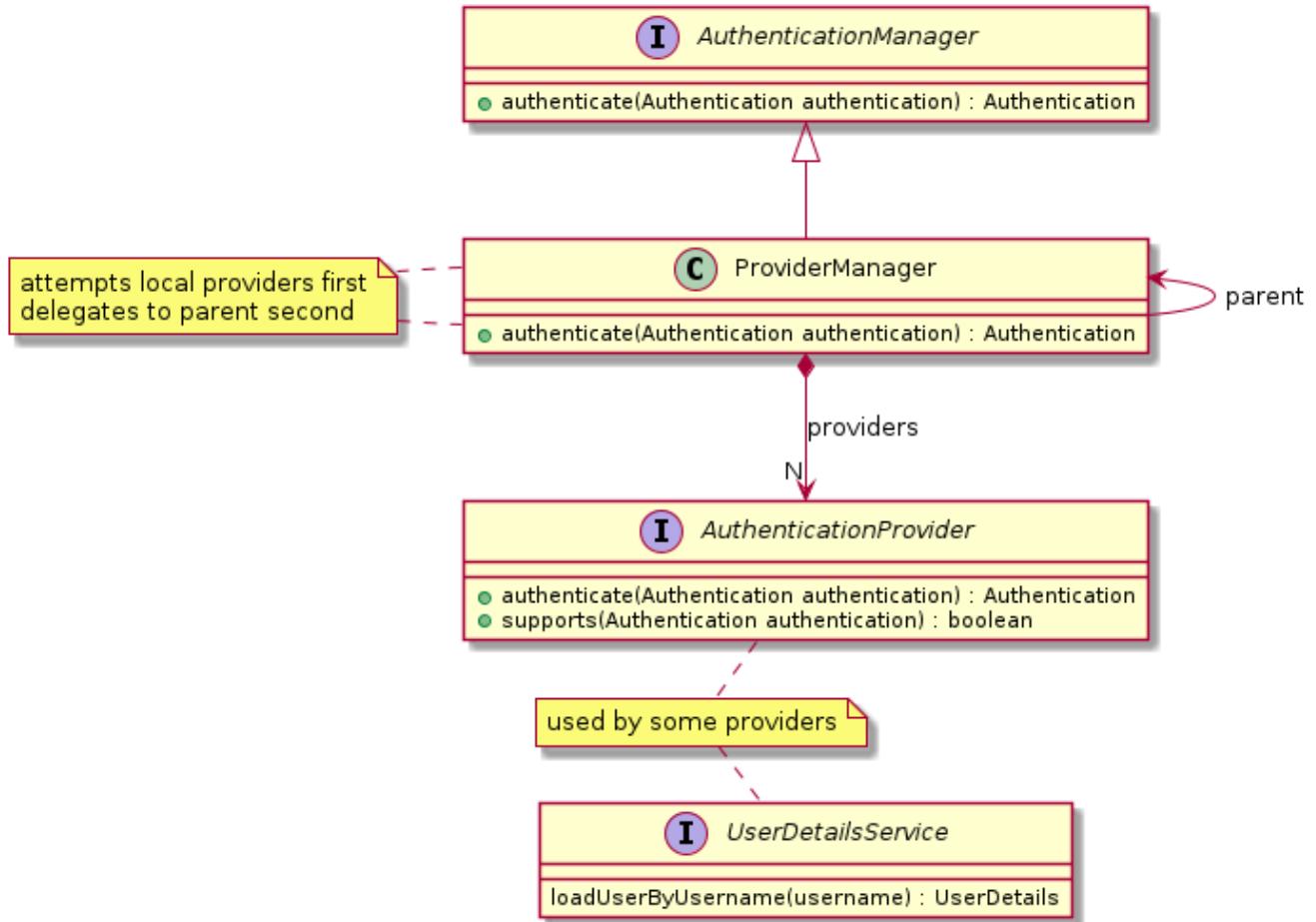


Figure 80. ProviderManager

198.2. AuthenticationManagerBuilder

It is the job of the `AuthenticationManagerBuilder` to assemble an `AuthenticationManager` with the required `AuthenticationProviders` and—where appropriate—`UserDetailsService`. The `AuthenticationManagerBuilder` is configured within the same `WebSecurityConfigurer` class we implemented to configure the `FilterChainProxy` and its `SecurityFilterChain` to trigger authentication.

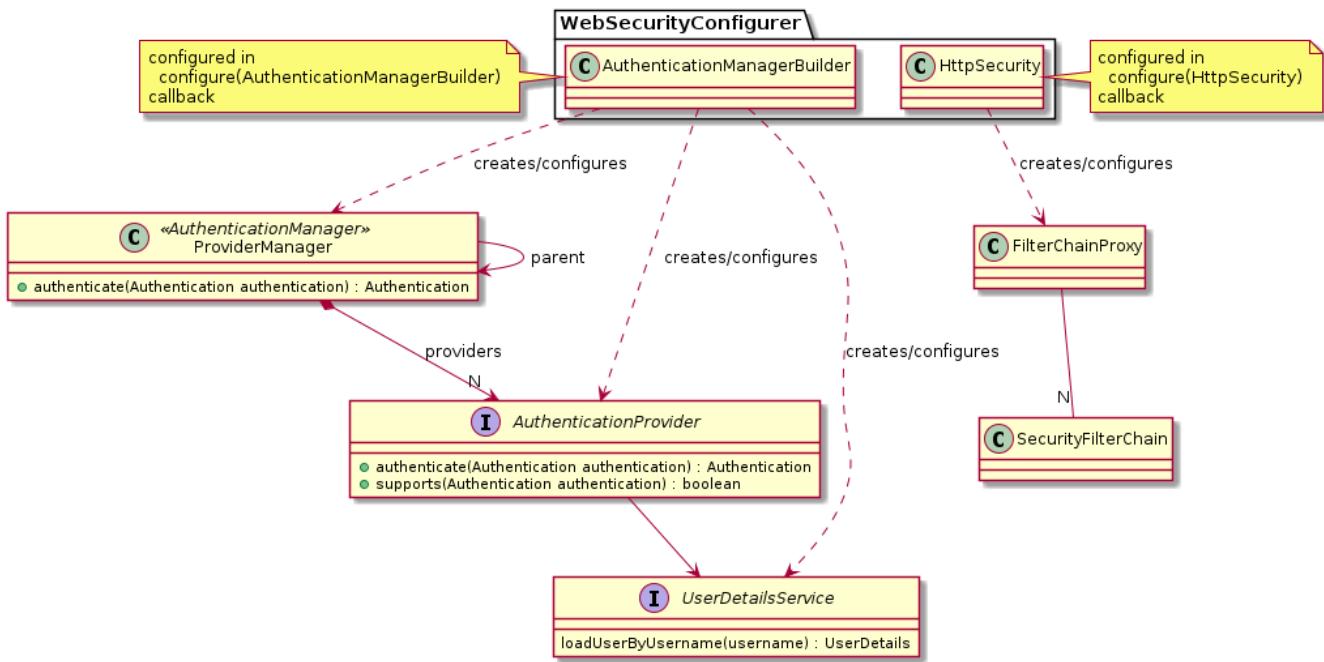


Figure 81. AuthenticationManagerBuilder

198.3. AuthenticationProvider

The **AuthenticationProvider** can answer two (2) questions:

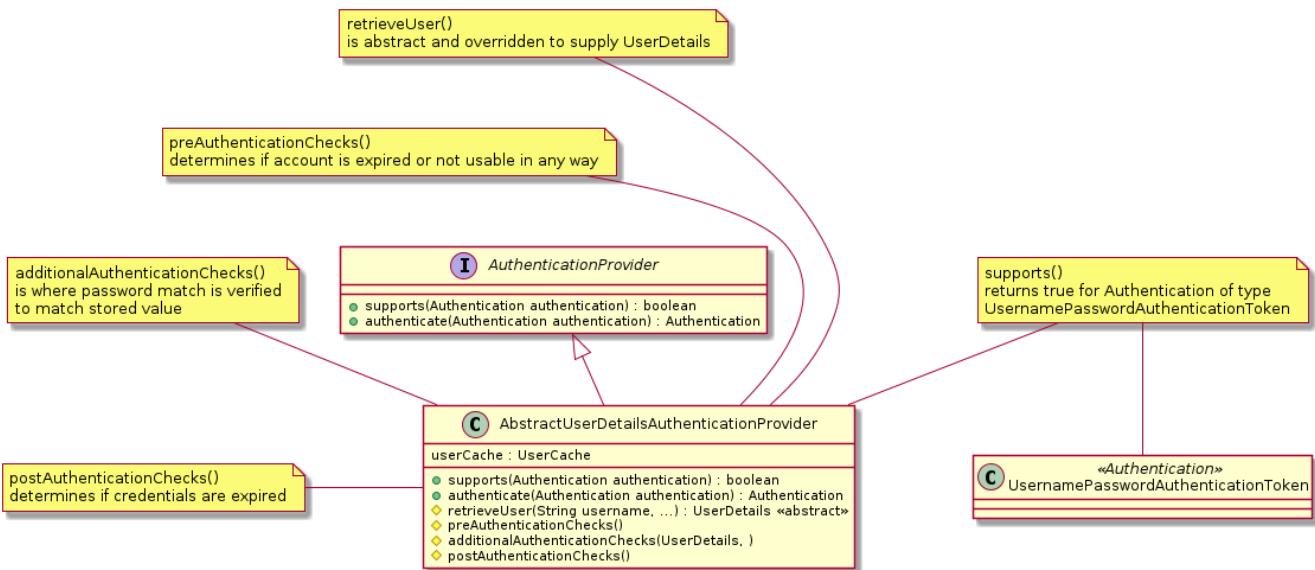
- do you support this type of authentication
- can you authenticate this attempt



198.4. 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



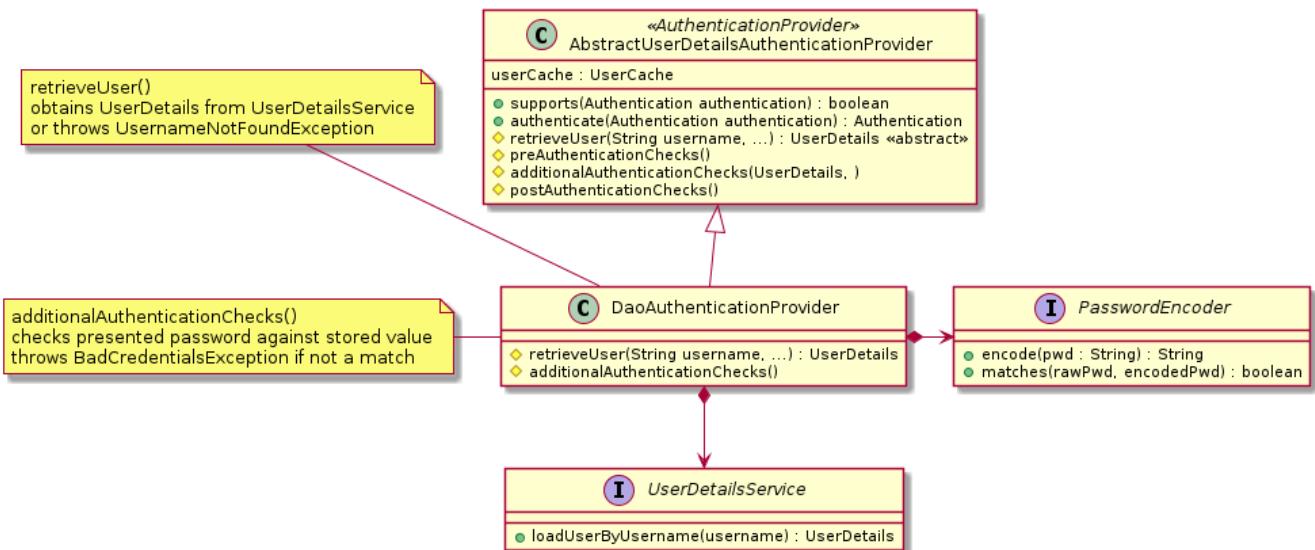
The instance will support any authentication token of type `UsernamePasswordAuthenticationToken` but will need at least two things:

- user details from storage
- a means to authenticate presented password

198.5. 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

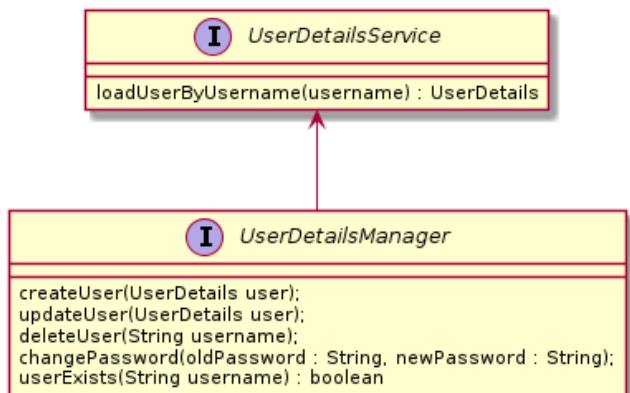


Now all we need is a `PasswordEncoder` and `UserDetailsService` to get all this rolling.

198.6. 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 199. WebSecurityConfigurer

At this point we know the framework of objects that need to be in place for authentication to complete. In this section we will learn how we can configure the assembly. To do that we will return to our `@Configuration` class that implements `WebSecurityConfigurer` and we have already setup our `FilterChainProxy` and `SecurityFilterChains`.

Continue Configuring Security APIConfiguration Class

```
@Configuration
public class APIConfiguration extends WebSecurityConfigurerAdapter {
    @Override
    public void configure(WebSecurity web) throws Exception { ... } ①
    @Override
    protected void configure(HttpSecurity http) throws Exception { ... } ②
    @Override
    protected void configure(AuthenticationManagerBuilder auth)
        throws Exception {
        auth//configure authentication ③
    }
}
```

① this is where we configured the top level matcher for the entire filter chain (e.g., to provide anonymous access to static content, isolate areas of resources)

② this is where we configured the filter chain (e.g., define authn requirements per URI pattern)

③ we will configure authentication implementation mechanisms in this builder configuration callback next

A basic `ProviderManager` configuration is provided for us and we can populate it with `AuthenticationProvider` instances using this builder. We can perform the population a number of different ways.

199.1. Directly Wire-up Parent AuthenticationManager

We can directly set the parent of this `AuthenticationManager` to another `AuthenticationManager` created elsewhere. Normally this will be injected from another portion of our application.

```

@Configuration
@Order(500)
@RequiredArgsConstructor
public class H2Configuration extends WebSecurityConfigurerAdapter {
    private final AuthenticationManager authenticationManager; ①

    @Override
    protected void configure(AuthenticationManagerBuilder auth)
        throws Exception {
        auth.parentAuthenticationManager(authenticationManager); ②
    }
}

```

① `AuthenticationManager` assembled elsewhere and injected in this `@Configuration` class

② injected `AuthenticationManager` to be the parent `AuthenticationManager` of what this builder builds

199.2. Directly Wire-up AuthenticationProvider

We can directly add a fully pre-assembled `AuthenticationProvider` to the builder. With this approach we have full freedom to implement any authentication.

Directly Wiring AuthenticationProvider into Builder

```

@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.authenticationProvider(/* fully pre-assembled provider */); ①
}

```

① directly setting a fully-assembled `AuthenticationProvider`

199.3. Use Local AuthenticationManagerBuilder

We can use the local builder methods 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.

Example InMemoryAuthentication Configuration

```
@Override  
protected void configure(AuthenticationManagerBuilder auth) throws Exception {  
    PasswordEncoder encoder = ...  
    auth.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

199.3.1. 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`.

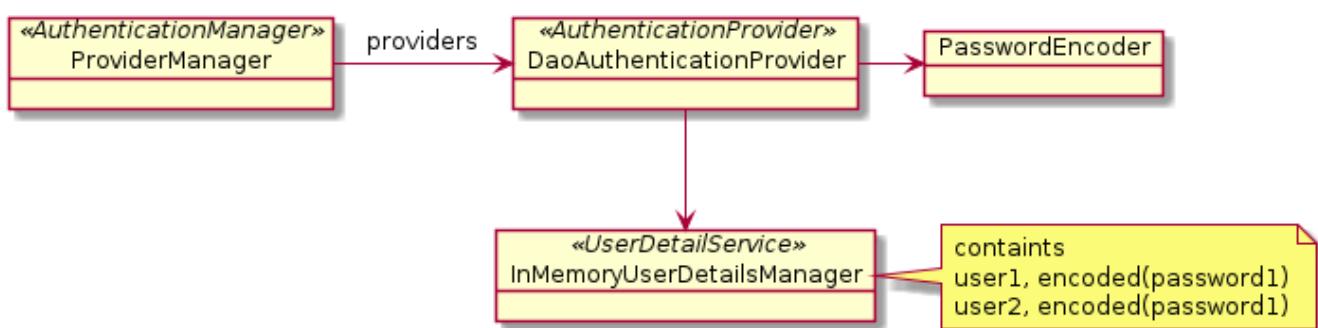


Figure 82. Example InMemoryUserDetailsManager

199.3.2. 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 \
-H "Authorization: BASIC dXNlcjE6cGFzc3dvcmQx" #user1:password1 ①
hello, jim :caller=user1

$ curl http://localhost:8080/api/authn/hello?name=jim \
-H "Authorization: BASIC dXNlcjI6cGFzc3dvcmQx" #user2:password1
hello, jim :caller=user2

$ curl http://localhost:8080/api/authn/hello?name=jim \
-H "Authorization: BASIC dXNlcjg6cGFzc3dvcmQ=" #userX:password
{"timestamp":"2020-07-12T07:31:11.895+00:00","status":401,
 "error":"Unauthorized","message":"Unauthorized","path":"/api/authn/hello"}
```

① #user1:password1 is an end of line comment telling us the contents of the base64 encoding and not part of the command

199.4. Define Service and Encoder @Bean

Another option in supplying a `UserDetailsService` is to define a globally accessible `UserDetailsService @Bean` to inject into our `WebSecurityConfigurer` class and use with the 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
    User.UserBuilder builder = User.builder().passwordEncoder(encoder::encode); ②
    List<UserDetails> users = List.of(
        builder.username("user1").password("password2").roles().build(), ③
        builder.username("user3").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

199.4.1. Inject UserDetailsService

Example Injected `UserDetailsService`

```
@Configuration
@Order(0)
@RequiredArgsConstructor
public class APIConfiguration extends WebSecurityConfigurerAdapter {
    private final UserDetailsService sharedUserDetailsService; ①

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.userDetailsService(sharedUserDetailsService); ②
    }
}
```

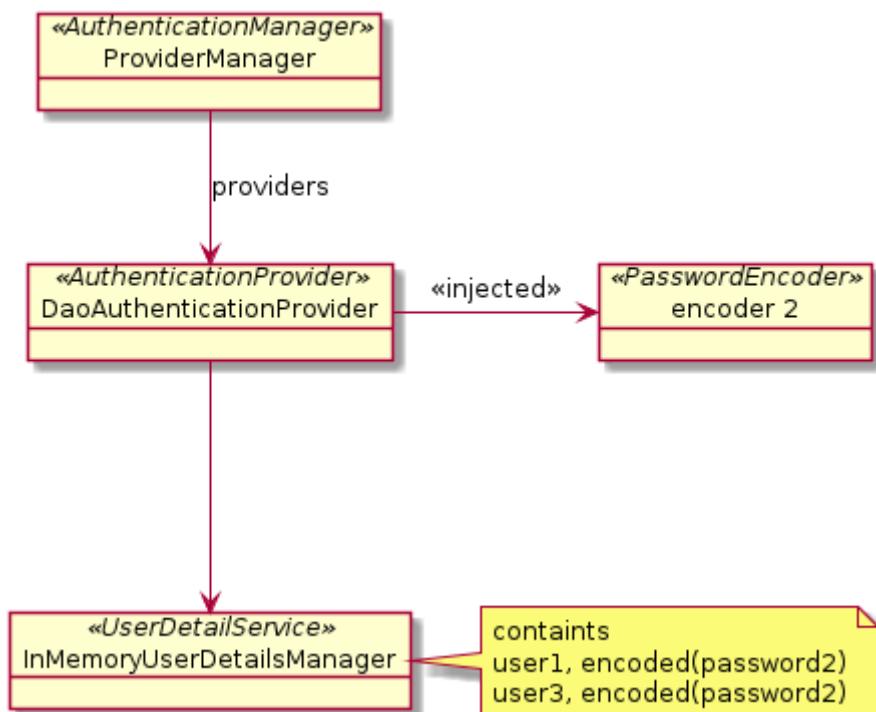
① injecting `UserDetailsService` into configuration class

② adding additional `UserDetailsService` to create additional `AuthenticationProvider`

199.4.2. Assembled Injected UserDetailsService

The results of the builder configuration are shown below where the builder assembled an `AuthenticationProvider` (`DaoAuthenticationProvider`) based on the injected `UserDetailsService` (`InMemoryUserDetailsManager`).

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`.



199.4.3. 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 \
-H "Authorization: BASIC dXNlcjE6cGFzc3dvcmQy" #user1:password2
hello, jim :caller=user1

$ curl http://localhost:8080/api/authn/hello?name=jim \
-H "Authorization: BASIC dXNlcjM6cGFzc3dvcmQy" #user3:password2
hello, jim :caller=user3

$ curl http://localhost:8080/api/authn/hello?name=jim \
-H "Authorization: BASIC dXNlcjg6cGFzc3dvcmQ=" #userX:password
{"timestamp":"2020-07-12T09:26:50.467+00:00", "status":401,
 "error": "Unauthorized", "message": "Unauthorized", "path": "/api/authn/hello"}
```

199.5. 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

```
@Configuration
@Order(0)
@RequiredArgsConstructor
public class APIConfiguration extends WebSecurityConfigurerAdapter {
    private final UserDetailsService sharedUserDetailsService; ②

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        PasswordEncoder encoder = ... ①
        auth.inMemoryAuthentication().passwordEncoder(encoder)
            .withUser("user1").password(encoder.encode("password1")).roles()
            .and()
            .withUser("user2").password(encoder.encode("password1")).roles();
        auth.userDetailsService(sharedUserDetailsService); ③
    }
}
```

① locally built `AuthenticationProvider` will use its own encoder

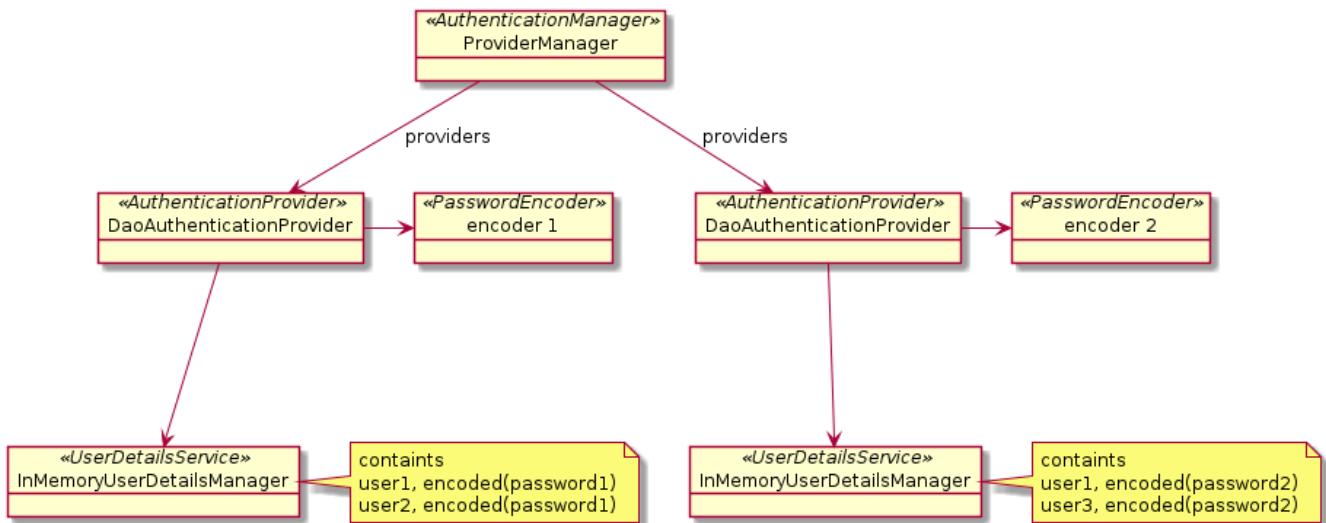
② `@Bean`-built `UserDetailsService` injected

③ injected `UserDetailsService` used to form second `AuthenticationProvider`

199.5.1. Assembled Combined AuthenticationProviders

The resulting `AuthenticationManager` ends up with two `AuthenticationProviders` that were assembled during the `configure()` method. Each `AuthenticationProviders` are

- implemented with the `DaoAuthenticationProvider` class
- make use of a `PasswordEncoder` and `UserDetailsService`



The left `UserDetailsService` was instantiated locally as an `InMemoryUserDetailsManager`, using a `@Bean` factory that instantiated the builder methods from the `InMemoryUserDetailsManager` directly. Since it was `AuthenticationManagerBuilder`. Since it was shared as a `@Bean`, the factory method used an locally built, it used its own choice of injected `PasswordEncoder` to assemble. `PasswordEncoder`

The two were brought together by our `configure()` method and now we have two sources of credentials to authenticate against.

199.5.2. 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 \
-H "Authorization: BASIC dXNlcjE6cGFzc3dvcmQx" #user1:password1
hello, jim :caller=user1

$ curl http://localhost:8080/api/authn/hello?name=jim \
-H "Authorization: BASIC dXNlcjE6cGFzc3dvcmQy" #user1:password2
hello, jim :caller=user1

$ curl http://localhost:8080/api/authn/hello?name=jim \
-H "Authorization: BASIC dXNlcjI6cGFzc3dvcmQx" #user2:password1
hello, jim :caller=user2

$ curl http://localhost:8080/api/authn/hello?name=jim \
-H "Authorization: BASIC dXNlcjM6cGFzc3dvcmQy" #user3:password2
hello, jim :caller=user3
```

199.6. AuthenticationManager @Bean

The `AuthenticationManager` that we worked so hard to construct can be made reusable to the other `WebSecurityConfigurers`. Those builders can inject this bean and set as its parent `AuthenticationManager` — as [shown earlier](#) earlier in this section.

AuthenticationManager @Bean

```
@Configuration
@Order(0)
@RequiredArgsConstructor
public class APIConfiguration extends WebSecurityConfigurerAdapter {
    private final UserDetailsService sharedUserDetailsService;
    private final UserDetailsService jdbcUserDetailsService;
    //...
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        //configure the builder
    }

    @Bean ①
    @Override //expose the built AuthenticationManager to be reused
    public AuthenticationManager authenticationManagerBean() throws Exception {
        return super.authenticationManagerBean(); ②
    }
}
```

① exposing the fully constructed `AuthenticationManager` as a `@Bean` to be injected/reused elsewhere

② use the return of `super.authenticationManagerBean()` to obtain an instance meant to be exposed as `@Bean`

Chapter 200. UserDetails

So now we know that all we need is to provide an `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 like we have been demonstrating.

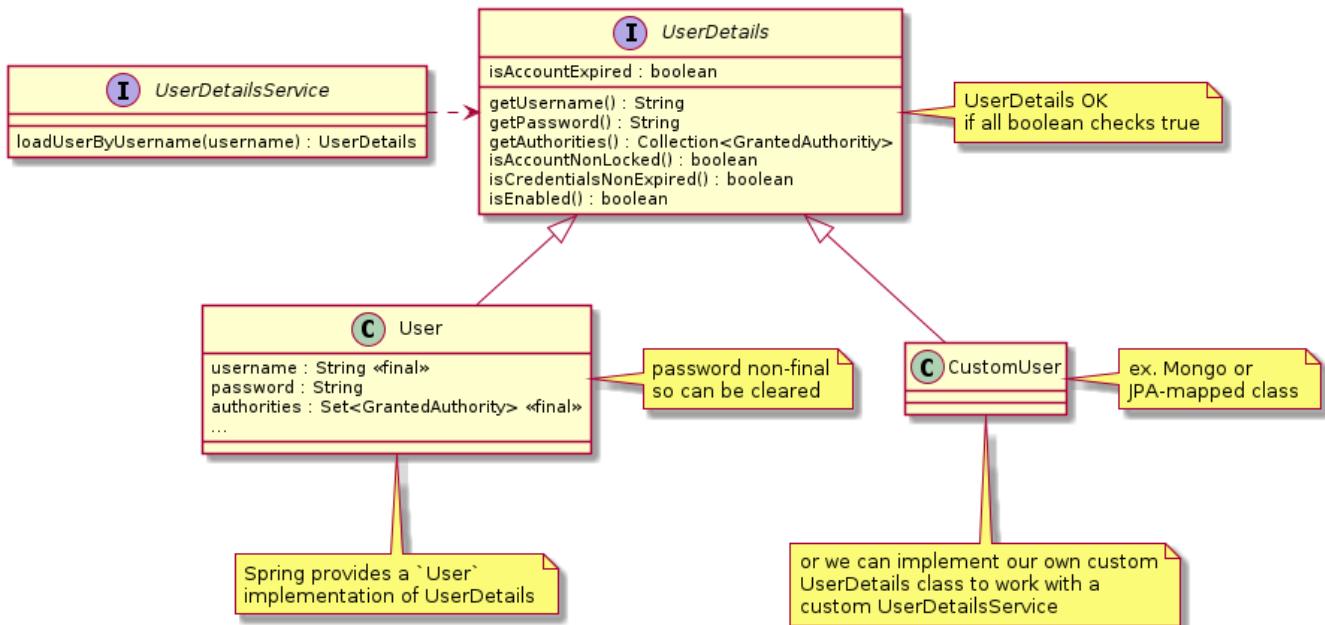


Figure 83. `UserDetailsService`

Chapter 201. 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.

201.1. 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.

201.2. BCryptPasswordEncoder

The `BCryptPasswordEncoder` uses a very strong Bcrypt algorithm and likely should be considered the default in production environments,

201.3. DelegatingPasswordEncoder

The `DelegatingPasswordEncoder` is a jack-of-all-encoders. It has one default way to encode but can match 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 202. 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 pretty 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.

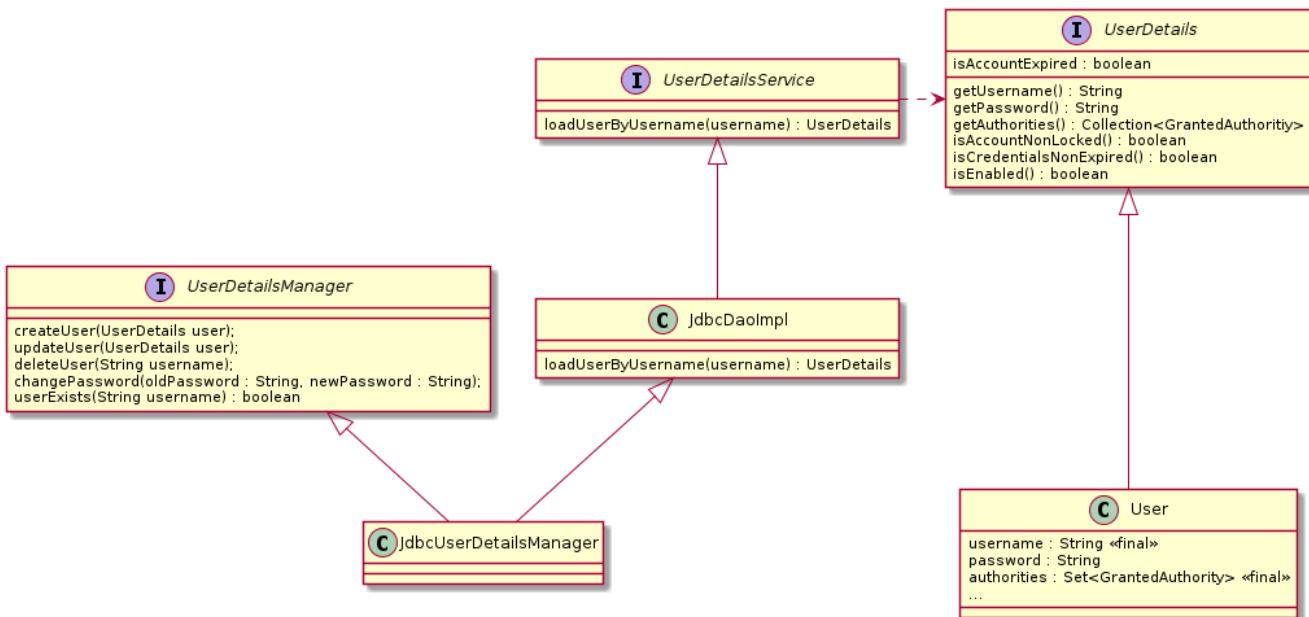


Figure 84. JDBC UserDetailsService

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`)
- (potentially inject and) add JDBC `UserDetailsService` to `AuthenticationManagerBuilder` in

`WebSecurityConfigurerAdapter` callback

202.1. 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.

202.2. 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>
```

202.3. 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

```
@Bean
public UserDetailsService jdbcUserDetailsService(DataSource dataSource) {
    JdbcDaoImpl jdbcUds = new JdbcDaoImpl();
    jdbcUds.setDataSource(dataSource);
    return jdbcUds;
}
```

202.4. Inject/Add JDBC UserDetailsService

From there, we can inject the JDBC `UserDetailsService`—like the in-memory version we injected earlier—into the API security `@Configuration` class and add it to the builder.

```
@Configuration
@Order(0)
@RequiredArgsConstructor
public class APIConfiguration extends WebSecurityConfigurerAdapter {
    private final UserDetailsService sharedUserDetailsService;
    private final UserDetailsService jdbcUserDetailsService; ①
    ...
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        PasswordEncoder encoder = NoOpPasswordEncoder.getInstance();
        auth.inMemoryAuthentication().passwordEncoder(encoder)
            .withUser("user1").password(encoder.encode("password1")).roles()
            .and()
            .withUser("user2").password(encoder.encode("password1")).roles();
        auth.userDetailsService(sharedUserDetailsService);
        auth.userDetailsService(jdbcUserDetailsService); ②
    }
}
```

① inject the JDBC `UserDetailsService` into `@Configuration` class

② add the additional `UserDetailsService` to the builder to form another `AuthenticationProvider`



Password Encoder can be explicitly set for AuthenticationProvider

The `userDetailsService()` method returns a builder that can accept an explicit `PasswordEncoder`

202.5. 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'
```

202.6. 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.

202.7. 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

H2 UI Security Configuration

```
@Configuration
@Order(500)
@RequiredArgsConstructor
public class H2Configuration extends WebSecurityConfigurerAdapter {
    private final AuthenticationManager authenticationManager; ①

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests(cfg->cfg ②
            .antMatchers("/login", "/logout").permitAll());
        http.authorizeRequests(cfg->cfg ③
            .antMatchers("/h2-console/**").authenticated());
        http.csrf(cfg->cfg.ignoringAntMatchers("/h2-console/**")); ④
        http.headers(cfg->cfg.frameOptions().disable()); ⑤
        http.formLogin().successForwardUrl("/h2-console"); ⑥
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.parentAuthenticationManager(authenticationManager);
    }
}
```

① injected `AuthenticationManager` bean exposed by `APIConfiguration`

② enable unauthenticated access to the login and logout URIs

③ require authenticated users by the application to reach the console

④ turn off CSRF only for the H2 console

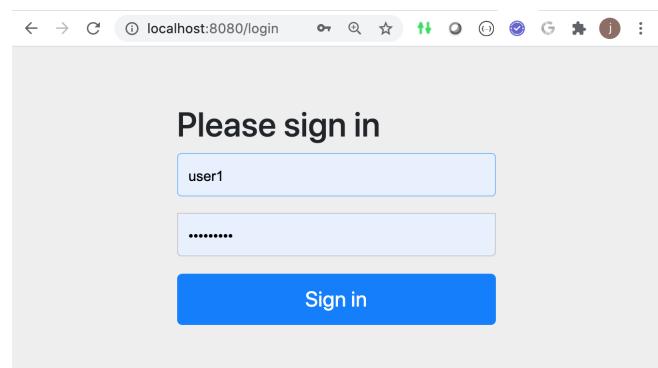
⑤ turn off display constraints for the H2 console

⑥ route successful logins to the H2 console

202.8. 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.

202.9. H2 Login

← → ⌂ ⓘ localhost:8080/h2-console/login.jsp?jsessionid=df4a0a5a0... ⌂ ⓘ

English Preferences Tools Help

Login

Saved Settings: Generic H2 (Embedded)

Setting Name: Generic H2 (Embedded) Save Remove

Driver Class: org.h2.Driver

JDBC URL: jdbc:h2:mem:users

User Name: sa

Password:

Connect Test Connection

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.

202.10. 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.

← → ⌂ ⓘ localhost:8080/h2-console/login.do?jsessionid=df4a0a5a013f0f... ⌂ ⓘ

Auto commit Max rows: 1000 Auto complete Off Auto select On ?

jdbc:h2:mem:users INFORMATION_SCHEMA Users H2 1.4.200 (2019-10-14)

Run Run Selected Auto complete Clear SQL statement:

Important Commands

?	Displays this Help Page
Ctrl+Enter	Shows the Command History
Shift+Enter	Executes the current SQL statement
Ctrl+Space	Executes the SQL statement defined by the text selection
Ctrl+Space	Auto complete
Alt+F4	Disconnects from the database

Sample SQL Script

202.11. 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.datasource.schema=classpath:database/users-schema.ddl
```

202.12. 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'
```

202.13. 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.datasource.schema=classpath:database/users-schema.ddl
spring.datasource.data=classpath:database/users-populate.sql
```

202.14. User DB Population

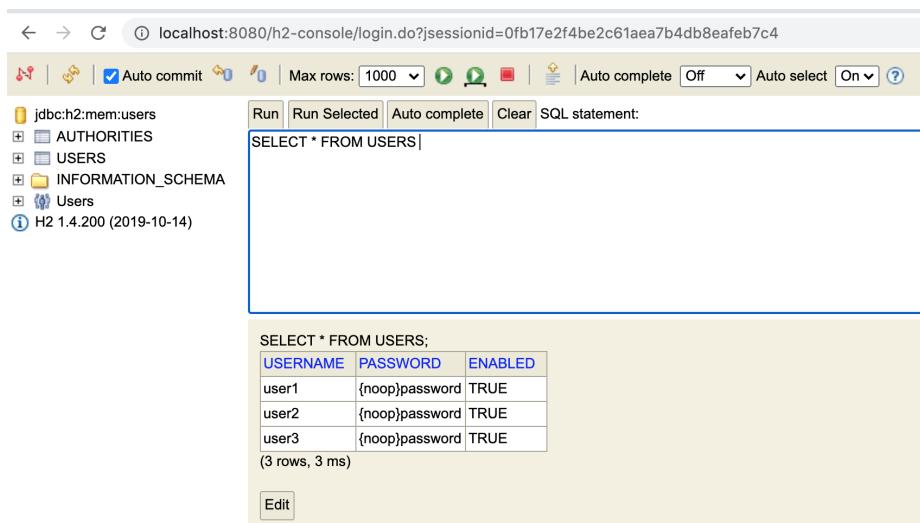
After the wave of schema commands has completed, the row population will take place filling the tables with our users, credentials, etc.

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'
```

202.15. 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 at localhost:8080/h2-console. The left sidebar shows database objects: `jdbc:h2:mem:users`, `AUTHORITIES`, `USERS`, `INFORMATION_SCHEMA`, `Users`, and the version `H2 1.4.200 (2019-10-14)`. The right panel has a toolbar with `Run`, `Run Selected`, `Auto complete`, `Clear`, and `SQL statement:`. Below it is a query input field containing `SELECT * FROM USERS;`. 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)

`Edit`

202.16. 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 \
-H "Authorization: BASIC dXNlcjE6cGFzc3dvcmQ=" #user1:password
hello, jim :caller=user1

$ curl http://localhost:8080/api/anonymous/hello?name=jim \
-H "Authorization: BASIC dXNlcjE6cGFzc3dvcmQx" #user1:password1
hello, jim :caller=user1

$ curl http://localhost:8080/api/anonymous/hello?name=jim \
-H "Authorization: BASIC dXNlcjE6cGFzc3dvcmQy" #user1:password2
hello, jim :caller=user1
```

However, we still have plaintext passwords in the database. Lets look to clean that up.

202.17. Encrypting Passwords

It would be bad practice to leave the user passwords in plaintext when we have the ability to store encrypted values. 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/YdcdP/Bq

$ htpasswd -bnBC 10 user2 password | cut -d\: -f2
$2y$10$9tYKBY7act5dN.2d7kumuOsHytIJW8i23Ua2Qogcm60M638IXMmLS

$ htpasswd -bnBC 10 user3 password | cut -d\: -f2
$2y$10$AH6uepcNasVx1Ye0hXX20.0X4cI3nXX.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

202.17.1. 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.9kuZ9jY9VYg/VylVTu88ZSCYy/YdcdP/Bq'
where username='user1';
update users
set password='{bcrypt}$2y$10$9tYKBY7act5dN.2d7kumuOsHytIJW8i23Ua2Qogcm6OM638IXMmLS'
where username='user2';
update users
set password='{bcrypt}$2y$10$AH6uepcNasVxlYeOhXX20.0X4cI3nXX.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.

202.17.2. 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 at `localhost:8080/h2-console/login.do?jsessionid=aebf4d31e86de779841f015da6325d74`. The left sidebar shows the database structure: `jdbc:h2:mem:users`, `AUTHORITIES`, `USERS`, `INFORMATION_SCHEMA`, and `Users`. The `Users` section indicates version `H2 1.4.200 (2019-10-14)`. The main area displays the results of the `SELECT * FROM USERS;` query:

USERNAME	PASSWORD	ENABLED
user1	{bcrypt}\$2y\$10\$UvKwrln7xPp35c5sbj.9kuZ9jY9VYg/VylVTu88ZSCYy/YdcdP/Bq	TRUE
user2	{bcrypt}\$2y\$10\$9tYKBY7act5dN.2d7kumuOsHytIJW8i23Ua2Qogcm6OM638IXMmLS	TRUE
user3	{bcrypt}\$2y\$10\$AH6uepcNasVxlYeOhXX20.0X4cI3nXX.LsicoDE5G6bCP34URZZF2	TRUE

(3 rows, 4 ms)

`Edit`

Figure 85. H2 User Access to Encrypted User Passwords

Chapter 203. Final Examples

203.1. 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

```
$ echo -n user1:password | base64 ①
dXNlcjE6cGFzc3dvcmQ=

$ curl http://localhost:8080/api/authn/hello?name=jim \
-H "Authorization: BASIC dXNlcjE6cGFzc3dvcmQ=" #user1:password ②
hello, jim :caller=user1

$ curl http://localhost:8080/api/authn/hello?name=jim \
-H "Authorization: BASIC dXNlcjI6cGFzc3dvcmQx" #user2:password1 ③
hello, jim :caller=user2

$ curl http://localhost:8080/api/authn/hello?name=jim \
-H "Authorization: BASIC dXNlcjM6cGFzc3dvcmQy" #user3:password2 ④
hello, jim :caller=user3
```

① we are still sending a base64 encoding of the plaintext password. The hash is currently created on the server-side before comparing to what is in the `UserDetailsService`

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

203.2. 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 /  
-H "Authorization: BASIC dXNlcjE6cGFzc3dvcmQ=" #user1:password ①  
hello, jim :caller=user1  
  
$ curl http://localhost:8080/api/authn/hello?name=jim /  
-H "Authorization: BASIC dXNlcjI6cGFzc3dvcmQ=" #user2:password ①  
hello, jim :caller=user2  
  
$ curl http://localhost:8080/api/authn/hello?name=jim /  
-H "Authorization: BASIC dXNlcjM6cGFzc3dvcmQ=" #user3:password ①  
hello, jim :caller=user3
```

① three separate user credentials stored in H2 database

Chapter 204. 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 © 2021 jim stafford (jim.stafford@jhu.edu)

Chapter 205. 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.

205.1. 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

205.2. 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 `AccessDecisionManager` and supply customized `AccessDecisionVoter` classes
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 206. 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 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" 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.

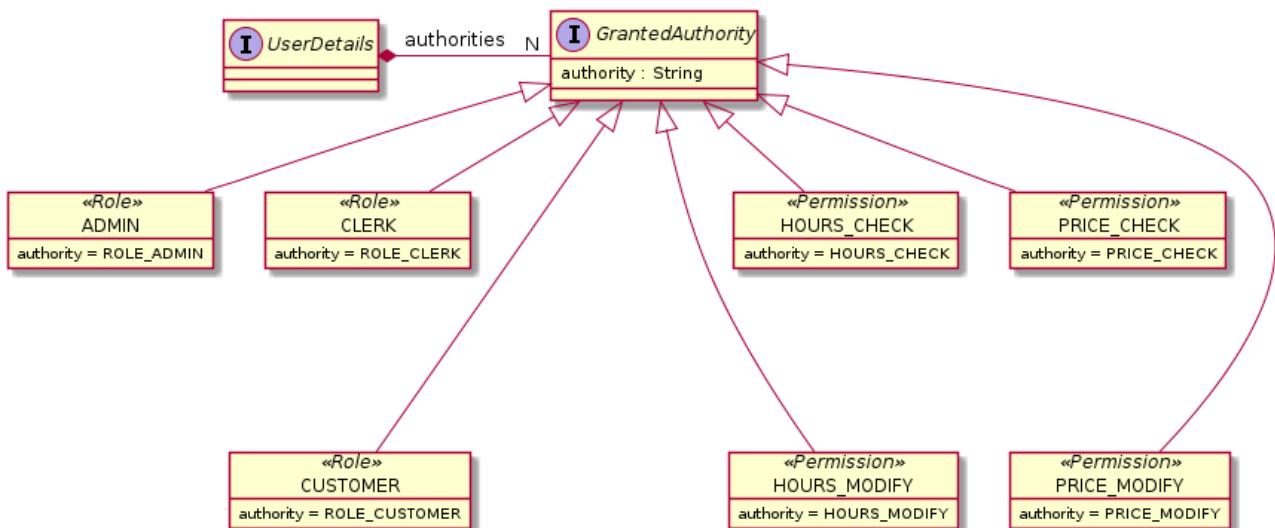


Figure 86. Role and Permission Authorities

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 look very similar but are not always treated equally.



Spring refers to authorities with **ROLE_** prefix as "roles" when the prefix is stripped away and anything with the raw value as "authorities". **ROLE_ADMIN** authority represents an **ADMIN** role. **PRICE_CHECK** permission is a **PRICE_CHECK** authority.

Chapter 207. Authorization Constraint Types

There are two primary ways we can express authorization constraints within Spring: path-based and annotation-based.

207.1. 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` configure callback we used with authentication.

Authn and Authz HttpSecurity Configuration

```
@Configuration
@Order(0)
@RequiredArgsConstructor
public 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.httpBasic(cfg->cfg.realmName("AuthzExample"));
        //remaining authn and upcoming authz goes here
    }
}
```

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.

Example 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

Out-of-the-box, path-based annotations support role inheritance, roles, and permission-based constraints. Path-based constraints also support [Spring Expression Language \(SpEL\)](#).

207.2. 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. Out-of-the-box, this annotation only supports roles and does not support role inheritance.

```
@Secured("ROLE_ADMIN") ①
@GetMapping(path = "admin", produces = {MediaType.TEXT_PLAIN_VALUE})
public ResponseEntity<String> doAdmin()
```

① ROLE_ prefix must be included in string

- 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. Out-of-the-box, this too only supports roles and does not support role inheritance.

```
@RolesAllowed("ROLE_ADMIN") ①
@GetMapping(path = "admin", produces = {MediaType.TEXT_PLAIN_VALUE})
public ResponseEntity<String> doAdmin()
```

① ROLE_ prefix must be included in string

- 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. It also provides support for role inheritance.

```
@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 208. 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.

208.1. 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) ?
            Arrays.asList(user.getUsername(), user.getAuthorities()) :
            Arrays.asList("null");
        String text = StringUtils.join(values);

        ResponseEntity<String> response = ResponseEntity.ok(text);
        return response;
    }
}
```

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 \
-H "Authorization: BASIC ZnJhc2llcjpwYXNzd29yZA==" #frasier:password
[frasier, [PRICE_CHECK, ROLE_CUSTOMER]]
```

208.2. 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
```

① frasier is assigned a (non-role) permission

208.3. Core FilterChainProxy Setup

The following shows the initial/core Security FilterProxyChain Setup carried over from earlier examples. We will add to this in a moment.

Core Security FilterChainProxy Setup

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.httpBasic(cfg->cfg.realmName("AuthzExample"));
    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.authorizeRequests(cfg->cfg.antMatchers(
        "/api/whoami",
        "/api/authorities/paths/anonymous/**")
        .permitAll());

    //more ...
}
```

208.4. 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 209. 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})
```

209.1. 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.

Example Path Role Authorization Constraints

```
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)

209.2. 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 \
-H "Authorization: BASIC c2FtOnBhc3N3b3Jk" #sam:password ①
[sam, [ROLE_ADMIN]]

$ curl http://localhost:8080/api/authorities/paths/clerk \
-H "Authorization: BASIC c2FtOnBhc3N3b3Jk" #sam:password ②
[sam, [ROLE_ADMIN]]

$ curl http://localhost:8080/api/authorities/paths/customer \
-H "Authorization: BASIC c2FtOnBhc3N3b3Jk" #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

209.3. 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 \
-H "Authorization: BASIC d29vZHk6cGFzc3dvcmQ=" #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 \
-H "Authorization: BASIC d29vZHk6cGFzc3dvcmQ=" #woody:password ②
[woody, [ROLE_CLERK]]

$ curl http://localhost:8080/api/authorities/paths/customer \
-H "Authorization: BASIC d29vZHk6cGFzc3dvcmQ=" #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 210. 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.

Path-based Authority Authorization Constraints

```
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`

210.1. 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 \  
-H "Authorization: BASIC bm9ybTpwYXNzd29yZA==" #norm:password ①  
[norm, [ROLE_CUSTOMER]]  
  
$ curl http://localhost:8080/api/authorities/paths/price \  
-H "Authorization: BASIC bm9ybTpwYXNzd29yZA==" #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

210.2. 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 \
-H "Authorization: BASIC ZnJhc2llcjpwYXNzd29yZA==" #frasier:password ①
[frasier, [PRICE_CHECK, ROLE_CUSTOMER]]
```



```
$ curl http://localhost:8080/api/authorities/paths/price \
-H "Authorization: BASIC ZnJhc2llcjpwYXNzd29yZA==" #frasier:password ②
[frasier, [PRICE_CHECK, ROLE_CUSTOMER]]
```

① `frazier` has the `CUSTOMER` role assigned required to call `customer` URI

② `frazier` has the `PRICE_CHECK` permission assigned required to call `GET price` URI

210.3. 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 \
-H "Authorization: BASIC c2FtOnBhc3N3b3Jk" #sam:password ①
[sam, [ROLE_ADMIN]]
```



```
$ curl http://localhost:8080/api/authorities/paths/price
-H "Authorization: BASIC d29vZHk6cGFzc3dvcmQ=" #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

210.4. 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.

Other Path Constraints

```
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

210.5. 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 \
-H "Authorization: BASIC ZnJhc2llcjpwYXNzd29yZA==" #frasier:password ①
[frasier, [PRICE_CHECK, ROLE_CUSTOMER]]\n\n$ curl http://localhost:8080/api/authorities/paths/nobody \
-H "Authorization: BASIC ZnJhc2llcjpwYXNzd29yZA==" #frasier:password ②
{"timestamp":"2020-07-14T18:09:38.669+00:00","status":403,
 "error":"Forbidden","message":"Forbidden","path":"/api/authorities/paths/nobody"}\n\n$ 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 `frazier` was able to access the `authn` URI because he was authenticated

② `frazier` 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 211. Authorization

With that example in place, we can look behind the scenes to see how this occurred.

211.1. 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.

The screenshot shows a Java debugger interface with the following details:

- Code View:** Lines 196 to 211 of the `doFilterInternal` method. The line with the breakpoint (line 199) is highlighted in blue. The condition `filters == null || filters.size() == 0` is shown as false. The code includes logging for trace and debug levels, and the creation of a `VirtualFilterChain`.
- Variables View:** A table showing local variables:
 - `request = {HttpServletRequest@6400}`
 - `response = {ResponseFacade@6401}`
 - `chain = {ApplicationFilterChain@6402}`
 - `firewallRequest = {StrictHttpFirewall$StrictFirewalledRequest@6403} "FirewalledRequest[org.apache.catalina.connector.RequestFacade@7b19f3af]"`
 - `firewallResponse = {FirewalledResponse@6404}`
 - `filters = {ArrayList@6405} size = 15
 - 0 = {WebAsyncManagerIntegrationFilter@7970}
 - 1 = {SecurityContextPersistenceFilter@7971}
 - 2 = {HeaderWriterFilter@7972}
 - 3 = {CsrfFilter@7973}
 - 4 = {LogoutFilter@7974}
 - 5 = {UsernamePasswordAuthenticationFilter@7975}
 - 6 = {DefaultLoginPageGeneratingFilter@7976}
 - 7 = {DefaultLogoutPageGeneratingFilter@7977}
 - 8 = {BasicAuthenticationFilter@7978}
 - 9 = {RequestCacheAwareFilter@7979}
 - 10 = {SecurityContextHolderAwareRequestFilter@7980}
 - 11 = {AnonymousAuthenticationFilter@7981}
 - 12 = {SessionManagementFilter@7982}
 - 13 = {ExceptionTranslationFilter@7983}
 - 14 = {FilterSecurityInterceptor@7984}`

A red arrow points from the text "Authz Filter" to the last element in the `filters` list, which is `14 = {FilterSecurityInterceptor@7984}`.

Figure 87. Review: FilterSecurityInterceptor At End of Chain

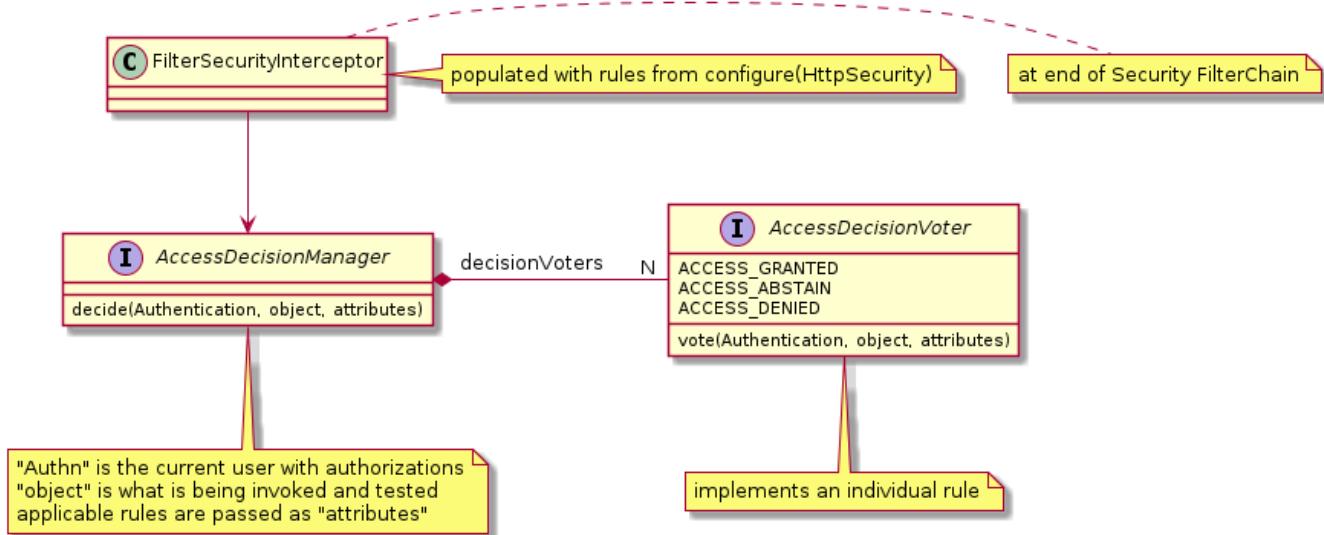
211.2. FilterSecurityInterceptor Calls

- the `FilterSecurityInterceptor` is at the end of the Security FilterChain and calls the `AccessDecisionManager` to decide whether the authenticated caller has access to the target object. The call quietly returns without an exception if accepted and throws an `AccessDeniedException` if denied.
- the assigned `AccessDecisionManager` is pre-populated with a set of `AccessDecisionVoters` (e.g.,

`WebExpressionVoter`) based on security definitions and passed the authenticated user, a reference to the target object, and the relevant rules associated with that target to potentially be used by the voters.

- the `AccessDecisionVoter` returns an answer that is either `ACCESS_GRANTED`, `ACCESS_ABSTAIN`, or `ACCESS_DENIED`.

The overall evaluation depends on the responses from the voters and the aggregate answer setting (e.g., affirmative, consensus, unanimous) of the manager.

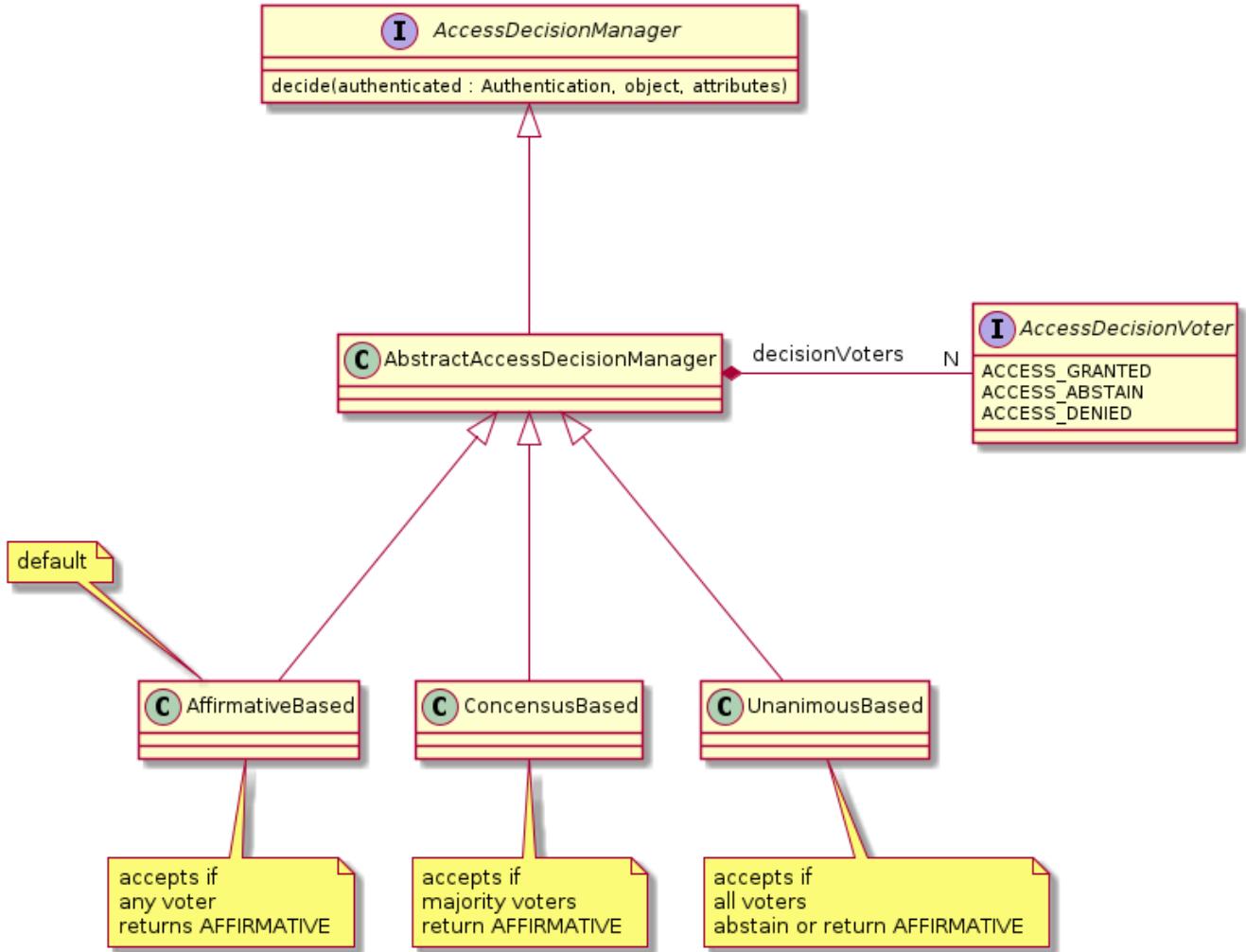


211.3. AccessDecisionManager

The `AccessDecisionManager` comes in three flavors and we can also create our own. The three flavors provided by Spring are

- `AffirmativeBased` - returns positive if any voter returns affirmative
- `ConsensusBase` - returns positive if majority of voters return affirmative
- `UnanimousBased` - returns positive if all voters return affirmative or abstain

Denial is signaled with a thrown `AccessDeniedException` exception. `AffirmativeBased` is the default. There is a setting in each for how to handle 100% abstain results — the default is access denied.



211.4. Assigning Custom AccessDecisionManager

The following code snippet shows an example of creating a `UnanimousBased` `AccessDecisionManager` and populating it with a custom list of voters.

Creating Custom AccessDecisionManager with Voters

```

@Bean
public AccessDecisionManager accessDecisionManager() {
    return new UnanimousBased(List.of(
        new WebExpressionVoter(),
        new RoleVoter(),
        new AuthenticatedVoter()));
}

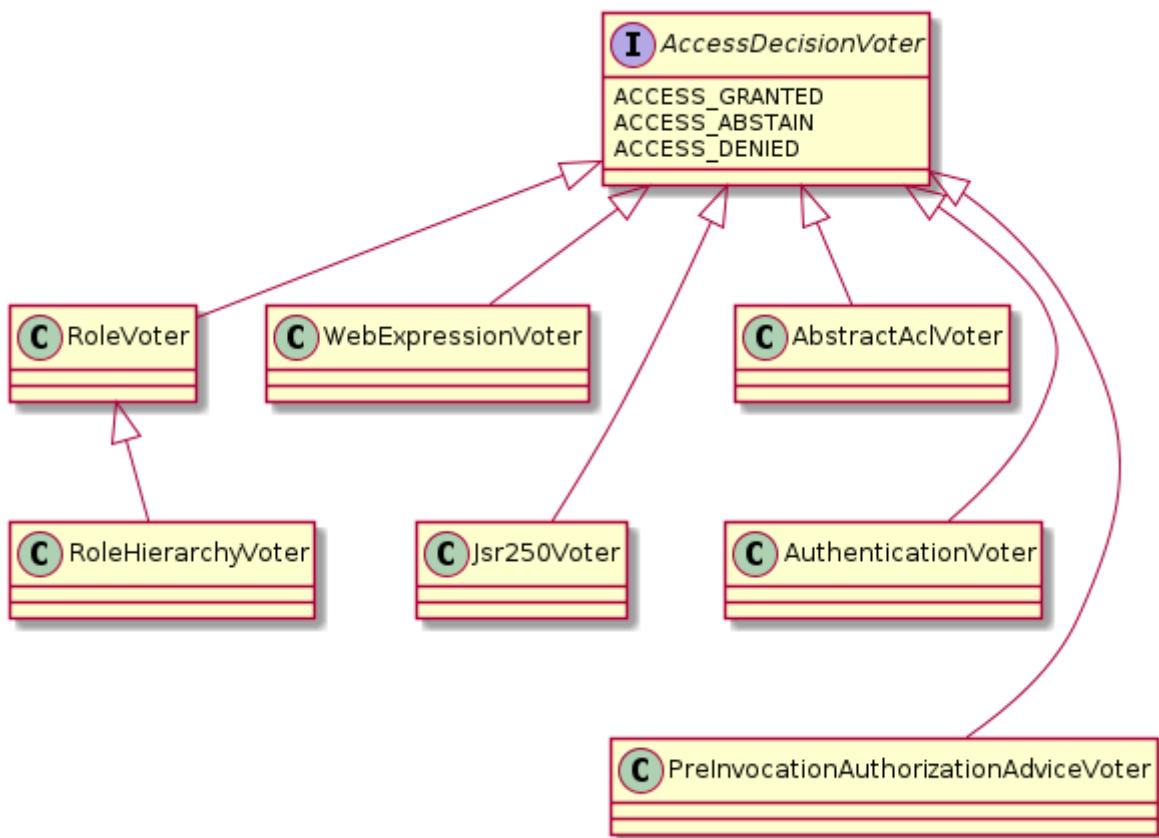
```

A custom `AccessDecisionManager` can be assigned to the builder returned from the access restrictions call.

```
http.authorizeRequests(cfg->cfg.antMatchers(
    "/api/authorities/paths/admin/**")
    .hasRole("ADMIN").accessDecisionManager(/* custom ADM here*/));
```

211.5. AccessDecisionVoter

There are several `AccessDecisionVoter` classes that take care of determining whether the specific constraints are satisfied, violated, or no determination. We can also create our own by extending or re-implementing any of the existing implementations and register using the technique shown in the snippets above.



In our first case, Spring converted our rules to be resolved to the `WebExpressionVoter`. Because of that—we will see many similarities to the constraint behavior of URI-based constraints and expression-based constraints covered towards the end of this lecture.

Chapter 212. Role Inheritance

Role inheritance provides an alternative to listing individual roles per URI constraint. Lets 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 \
-H "Authorization: BASIC c2FtOnBhc3N3b3Jk" #sam:password
{"timestamp":"2020-07-14T20:15:19.931+00:00","status":403,"error":"Forbidden",
 "message":"Forbidden","path":"/api/authorities/paths/customer"}
```

212.1. 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() {
    RoleHierarchyImpl roleHierarchy = new RoleHierarchyImpl();
    roleHierarchy.setHierarchy(StringUtils.join(Arrays.asList(
        "ROLE_ADMIN > ROLE_CLERK", ①
        "ROLE_CLERK > ROLE_CUSTOMER"), ②
        System.lineSeparator())); ③
    return roleHierarchy;
}
```

① role `ADMIN` will inherit all accessed applied to role `CLERK`

② role `CLERK` will inherit all accessed applied to role `CUSTOMER`

③ String expression built using new lines

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 \
-H "Authorization: BASIC c2FtOnBhc3N3b3Jk" #sam:password
[sam, [ROLE_ADMIN]]
```

Chapter 213. @Secure

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.

213.1. Enabling @Secured Annotations

@Secured annotations are disabled by default. We can enable them by supplying a `@EnableGlobalMethodSecurity` annotation with `securedEnabled` set to true.

Enabling @Secured

```
@Configuration
@EnableGlobalMethodSecurity(
    securedEnabled = true // @Secured({"ROLE_MEMBER"})
)
@RequiredArgsConstructor
public class SecurityConfiguration {
```

213.2. @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

213.3. @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 \
-H "Authorization: BASIC c2FtOnBhc3N3b3Jk" #sam:password ①
[sam, [ROLE_ADMIN]]
$ curl http://localhost:8080/api/authorities/secured/admin \
-H "Authorization: BASIC d29vZHk6cGFzc3dvcmQ=" #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

213.4. @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 \
-H "Authorization: BASIC d29vZHk6cGFzc3dvcmQ=" #woody:password
[woody, [ROLE_CLERK]]

$ curl http://localhost:8080/api/authorities/secured/price \
-H "Authorization: BASIC c2FtOnBhc3N3b3Jk" #sam:password
[sam, [ROLE_ADMIN]]
```

213.5. @Secured Only Processing Roles

However, **@Secured** evaluates using a **RoleVoter**, which only processes roles.

Attempt to use @Secured Annotation for Permissions

```
@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 ignored

Therefore, we cannot assign a **@Secured** to allow a permission like we did with the URI constraint.

@Secured Annotation only Supporting Roles

```
$ curl http://localhost:8080/api/authorities/paths/price \
-H "Authorization: BASIC ZnJhc2llcjpwYXNzd29yZA==" #frasier:password ①
[frasier, [PRICE_CHECK, ROLE_CUSTOMER]]

$ curl http://localhost:8080/api/authorities/secured/price \
-H "Authorization: BASIC ZnJhc2llcjpwYXNzd29yZA==" #frasier:password ②
{"timestamp": "2020-07-14T21:24:20.665+00:00", "status": 403,
 "error": "Forbidden", "trace": "org.springframework.security.access.AccessDeniedException
 ...(lots!!!)"}
```

① frasier can call URI **GET paths/price** because he has permission **PRICE_CHECK** and URI-based constraints honor non-role authorities (i.e., permissions)

② frasier cannot call URI **GET secured/price** because **checkPrice()** is constrained by **@Secured** and that only supports roles

213.6. @Secured Does Not Support Role Inheritance

@Secured annotation does not appear to support role inheritance we put in place when securing URIs.

```
$ curl http://localhost:8080/api/authorities/paths/clerk \
-H "Authorization: BASIC c2FtOnBhc3N3b3Jk" #sam:password ①
[sam, [ROLE_ADMIN]]  
  
$ curl http://localhost:8080/api/authorities/secured/clerk \
-H "Authorization: BASIC c2FtOnBhc3N3b3Jk" #sam:password ②
{"timestamp":"2020-07-14T21:48:40.063+00:00","status":403,
 "error":"Forbidden","trace":"org.springframework.security.access.AccessDeniedException
 ...(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

Chapter 214. 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 a 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 \
-H "Authorization: BASIC c2FtOnBhc3N3b3Jk" #sam:password ②
>{"timestamp":"2020-07-14T21:48:40.063+00:00","status":403,
"error":"Forbidden","trace":"org.springframework.security.access.AccessDeniedException
...(lots!!!)
```

214.1. 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

214.2. `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 thru `@ExceptionAdvice`

```
$ curl http://localhost:8080/api/authorities/secured/clerk \
-H "Authorization: BASIC c2FtOnBhc3N3b3Jk" #sam:password
{"url":"http://localhost:8080/api/authorities/secured/clerk","message":"Forbidden",
"description":"caller[sam] is forbidden from making this request",
"date":"2020-07-14T21:56:32.743996Z"}
```

Chapter 215. 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.

215.1. Enabling JSR-250

JSR-250 authorization annotations are also disabled by default. We can enable them the same as `@Secured` by setting the `@EnableGlobalMethodSecurity.jsr250Enabled` value to true.

Enabling JSR-250

```
@Configuration
@EnableGlobalMethodSecurity(
    jsr250Enabled = true // @RolesAllowed({"ROLE_MANAGER"})
)
@RequiredArgsConstructor
public class SecurityConfiguration {
```

215.2. `@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 `ROLE_ADMIN` authority expression.

Example `@RolesAllowed` Annotation

```
@RestController
@RequestMapping("/api/authorities/jsr250")
@RequiredArgsConstructor
public class Jsr250AuthoritiesController {
    private final WhoAmIController whoAmI;

    @RolesAllowed("ROLE_ADMIN") ①
    @GetMapping(path = "admin", produces = {MediaType.TEXT_PLAIN_VALUE})
    public ResponseEntity<String> doAdmin(
        @AuthenticationPrincipal UserDetails user) {
        return whoAmI.getCallerInfo(user);
    }
}
```

① role is expressed with `ROLE_` prefix

215.3. `@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 \
-H "Authorization: BASIC c2FtOnBhc3N3b3Jk" #sam:password ①
[sam, [ROLE_ADMIN]]\n\n$ curl http://localhost:8080/api/authorities/jsr250/admin \
-H "Authorization: BASIC d29vZHk6cGFzc3dvcmQ=" #woody:password ②
{"url":"http://localhost:8080/api/authorities/jsr250/admin","message":"Forbidden",
"description":"caller[woody] is forbidden from making this request",
"date":"2020-07-14T22:10:31.177471Z"}
```

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

215.4. 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()
```

215.5. 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 \
-H "Authorization: BASIC c2FtOnBhc3N3b3Jk" #sam:password ①
[sam, [ROLE_ADMIN]]\n\n$ curl http://localhost:8080/api/authorities/jsr250/price \
-H "Authorization: BASIC d29vZHk6cGFzc3dvcmQ=" #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

215.6. JSR-250 Does not Support Non-Role Authorities

Out-of-the-box, JSR-250 authorization annotation processing does not support non-Role authorizations. The following example shows where `frazer` 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 \
-H "Authorization: BASIC ZnJhc2llcjpwYXNzd29yZA==" #frasier:password ①
[frasier, [PRICE_CHECK, ROLE_CUSTOMER]]\n\n$ curl http://localhost:8080/api/authorities/jsr250/price \
-H "Authorization: BASIC ZnJhc2llcjpwYXNzd29yZA==" #frasier:password ②
{"url":"http://localhost:8080/api/authorities/jsr250/price","message":"Forbidden",
"description":"caller[frasier] is forbidden from making this request",
"date":"2020-07-14T22:13:26.247328Z"}
```

① `frasier` can invoke URI `GET paths/price` because he has the `PRICE_CHECK` authority and URI-based constraints support non-role authorities

② `frazier` cannot invoke JSR-250 constrained `checkPrice()` even though he has `PRICE_CHECK` permission because JSR-250 does not support non-role authorities

Chapter 216. Expressions

As demonstrated, `@Secured` and JSR-250-based 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 adding the following setting to the `@EnableGlobalMethodSecurity` annotation.

Enable Expressions

```
@EnableGlobalMethodSecurity(  
    prePostEnabled = true // @PreAuthorize("hasAuthority('ROLE_ADMIN')"),  
    @PreAuthorize("hasRole('ADMIN')")  
)
```

216.1. 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

216.2. 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 \
-H "Authorization: BASIC c2FtOnBhc3N3b3Jk" #sam:password ①
[sam, [ROLE_ADMIN]]  
  
$ curl http://localhost:8080/api/authorities/expressions/admin \
-H "Authorization: BASIC d29vZHk6cGFzc3dvcmQ=" #woody:password ②
{"url":"http://localhost:8080/api/authorities/expressions/admin","message":"Forbidden"
,
"description":"caller[woody] is forbidden from making this request",
"date":"2020-07-14T22:31:07.669546Z"}
```

① sam can invoke `doAdmin()` because he has the `ADMIN` role

② woody cannot invoke `doAdmin()` because he does not have the `ADMIN` role

216.3. Expressions Support Permissions and Role Inheritance

As noted earlier with URI-based constraints, exceptions support non-role authorities and role inheritance. 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 \
-H "Authorization: BASIC c2FtOnBhc3N3b3Jk" #sam:password
[sam, [ROLE_ADMIN]]
```

216.4. 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 \
-H "Authorization: BASIC c2FtOnBhc3N3b3Jk" #sam:password ①
[sam, [ROLE_ADMIN]]

$ curl http://localhost:8080/api/authorities/expressions/price \
-H "Authorization: BASIC d29vZHk6cGFzc3dvcmQ=" #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 \
-H "Authorization: BASIC ZnJhc2llcjpwYXNzd29yZA==" #frasier:password ①
[frasier, [PRICE_CHECK, ROLE_CUSTOMER]]
```

- ① frazier 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 \
-H "Authorization: BASIC bm9ybTpwYXNzd29yZA==" #norm:password ①
[norm, [ROLE_CUSTOMER]]

$ curl http://localhost:8080/api/authorities/expressions/price \
-H "Authorization: BASIC bm9ybTpwYXNzd29yZA==" #norm:password ②
{"url": "http://localhost:8080/api/authorities/expressions/price", "message": "Forbidden",
 "description": "caller[norm] is forbidden from making this request",
 "date": "2020-07-14T22:48:04.771588Z"}
```

- 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 217. 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 an `AccessDecisionManager` and `AccessDecisionVoter` 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

JWT/JWS Token Authn/Authz

copyright © 2021 jim stafford (jim.stafford@jhu.edu)

Chapter 218. 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.

218.1. 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`

218.2. 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 219. 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.

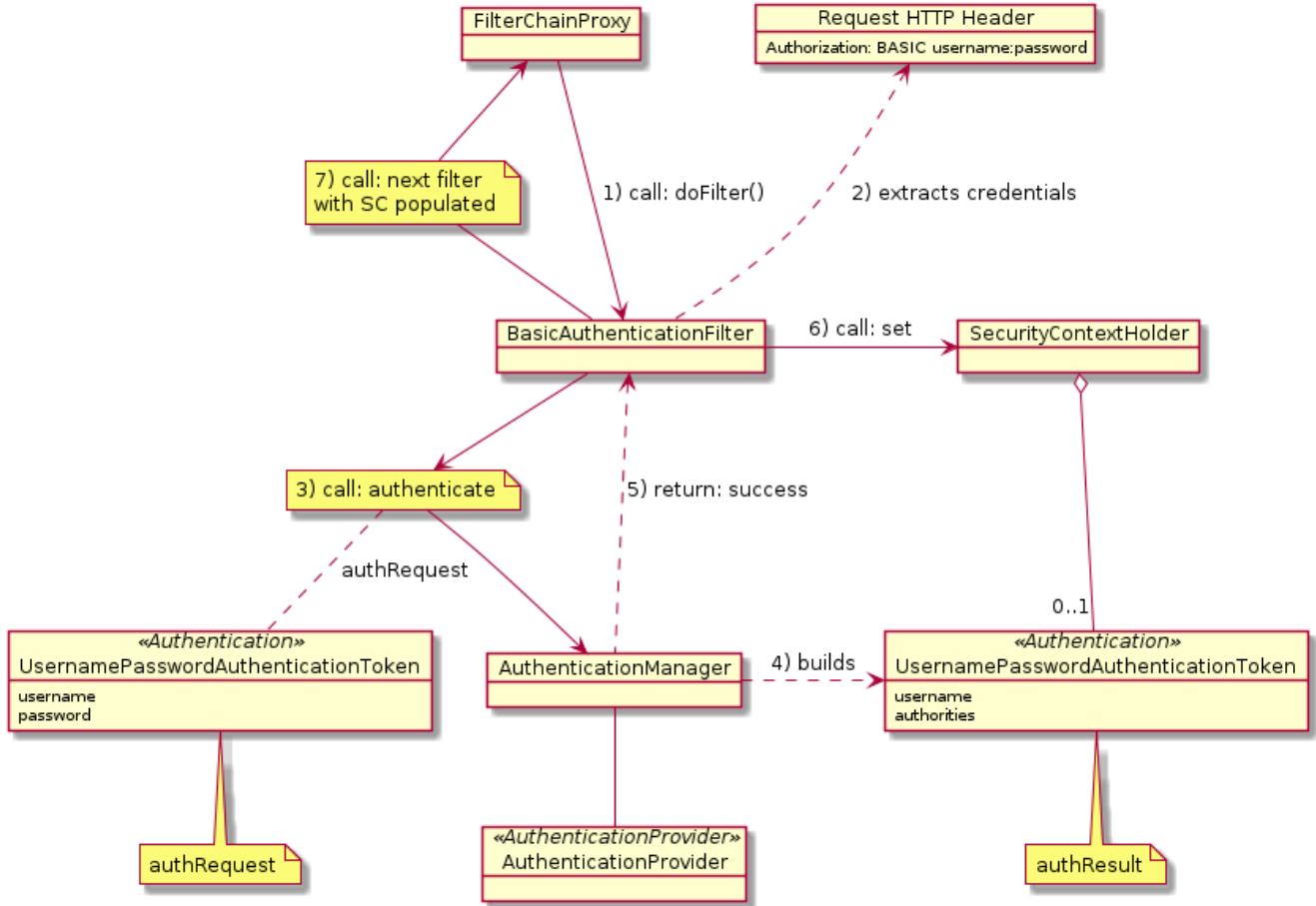
The benefit to BASIC is that it is stateless and can work with multiple servers—whether clustered or peer services. The bad part about BASIC is that we must present the credentials each time and the services must have access to our user details (including passwords) to be able to do anything with them.

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.

219.1. 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 220. Tokens

With token authentication, we are going to break the flow into two parts: authentication/login and authorization/operation.

220.1. 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".

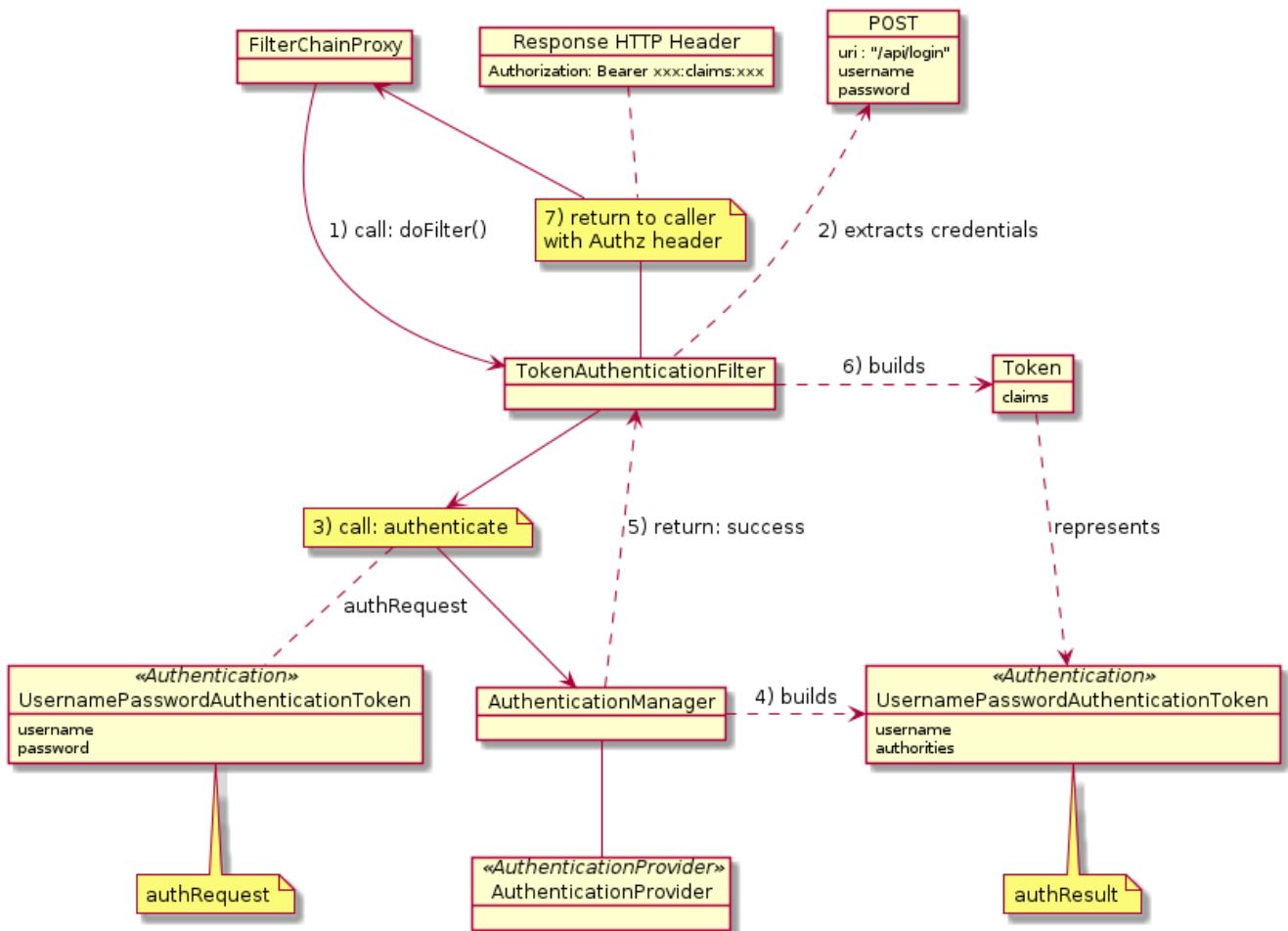


Figure 88. Example Notional Token Authentication/Login

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.

220.2. 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.

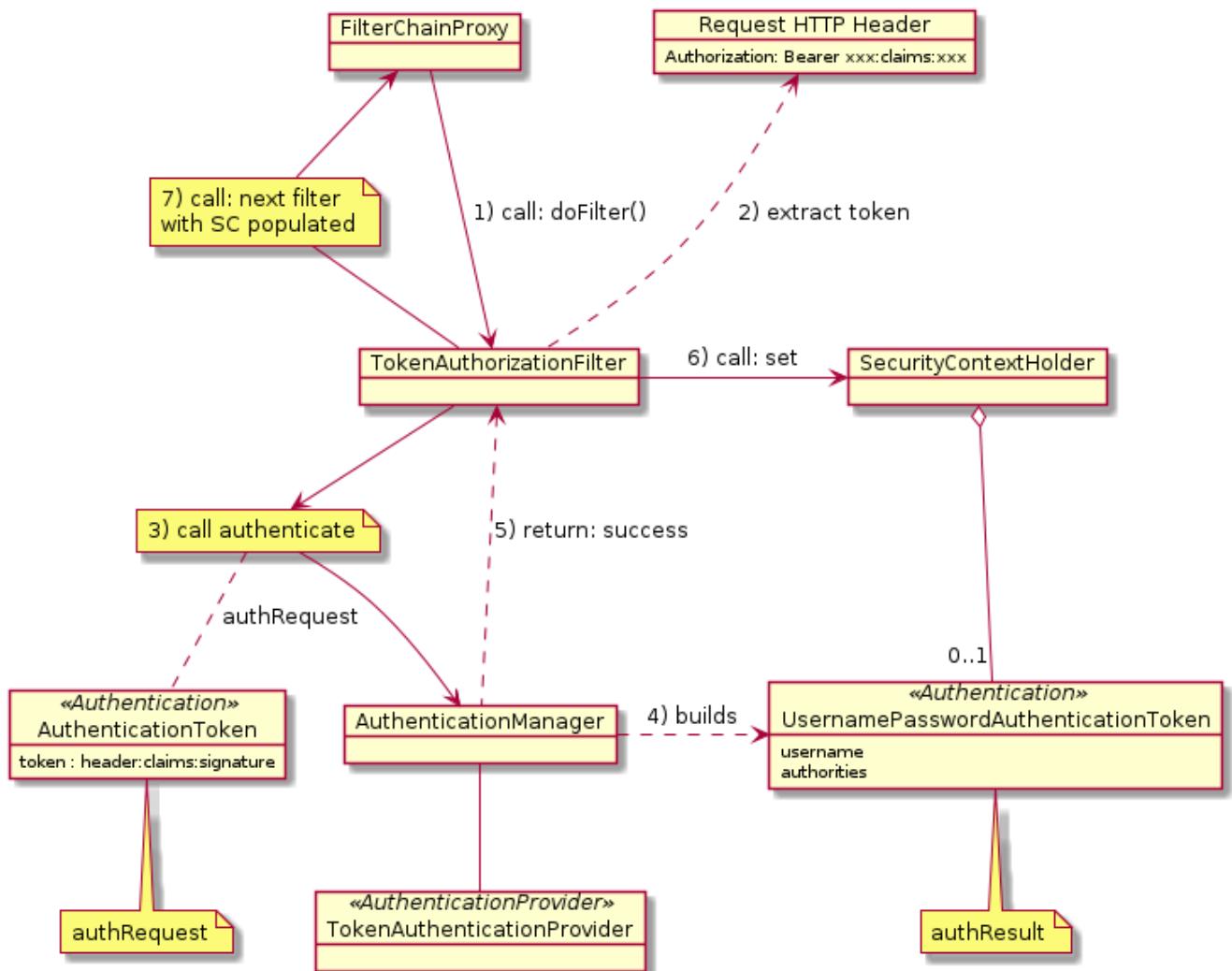


Figure 89. Example Notational Token Authorization/Operation

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.

220.3. 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.

220.4. JWT Terms

The following table contains some key, introductory terms related to JWT.

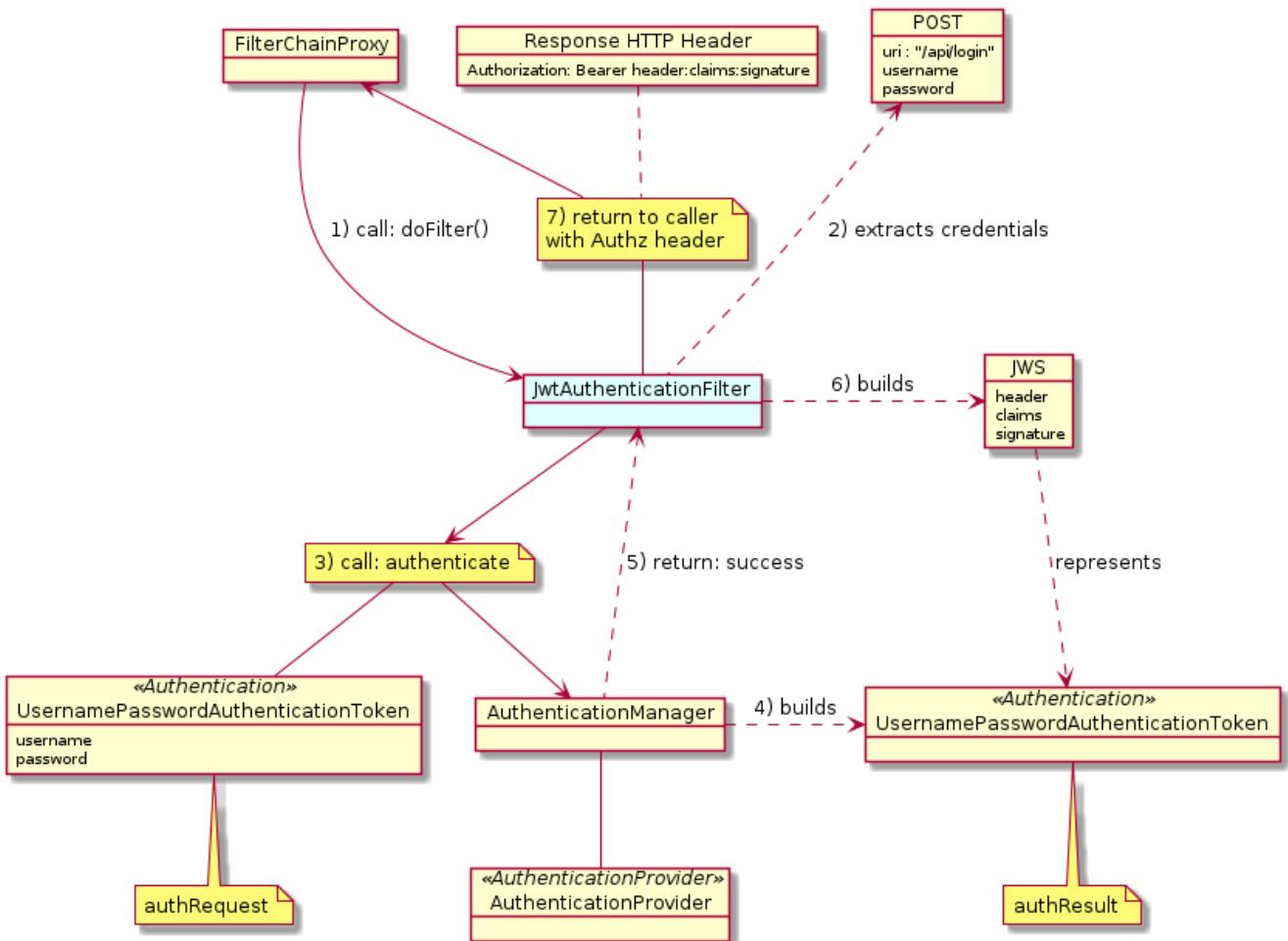
JSON Web Token (JWT)	a compact JSON claims representation that makes up the payload of a JWS or JWE structure e.g., <code>{"sub": "user1", "auth": ["ROLE_ADMIN"]}</code> . The JSON document is referred to as the JWT Claim Set. Basically — this is where we place what we want to represent. In our case, 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 ("alg":"none")
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.

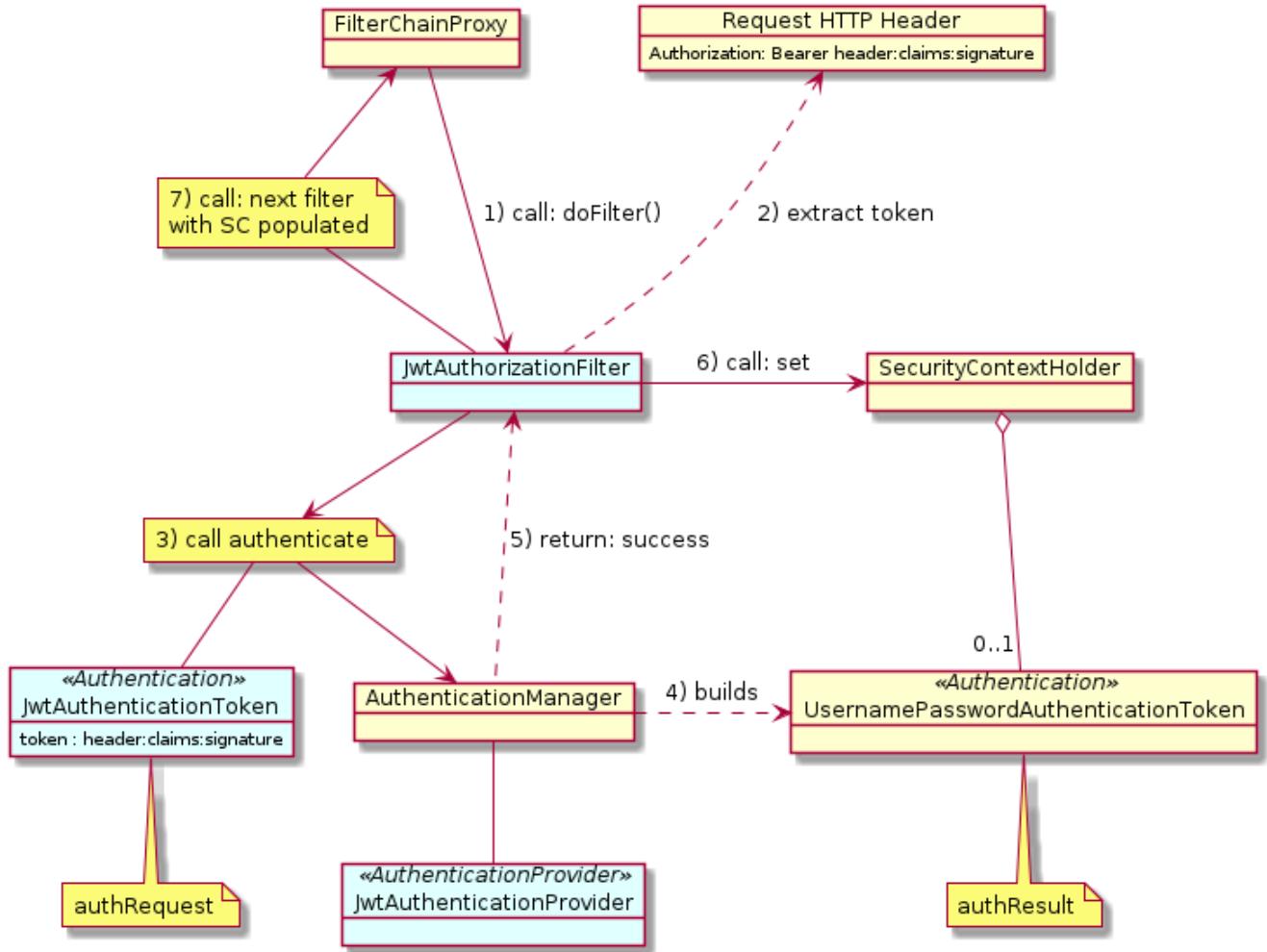
Chapter 221. JWT Authentication

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.

221.1. Example JWT Authentication/Login Flow



221.2. 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 222. 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 223. 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

223.1. 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 224. 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;
```

224.1. 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.

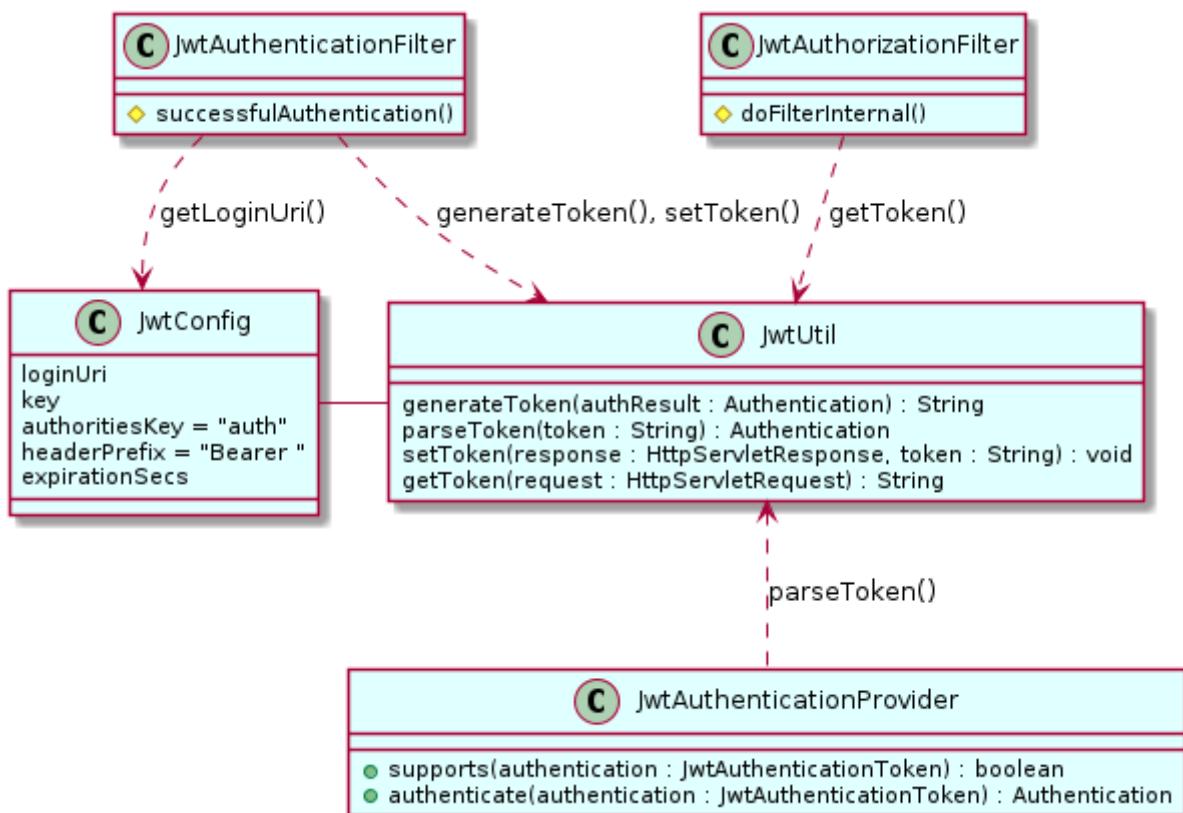


Figure 90. Dependencies on `JwtUtil`

224.2. 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

224.3. 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

224.4. 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
HAIoJE10TQ40Tk1MTcsImF1dGhvcmI0aWVzIjpBI1BSSUNFX0NIRUNLIiwiUk9MRV9DVVNUT01FUijdLCJqdGk
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.

224.5. 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.

224.6. 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.

224.7. 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 225. 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;
```

225.1. JwtAuthenticationFilter Relationships

The `JwtAuthenticationFilter` fills out the abstract workflow of the `AbstractAuthenticationProcessingFilter` by implementing two primary methods: `attemptAuthentication()` and `successfulAuthentication()`.

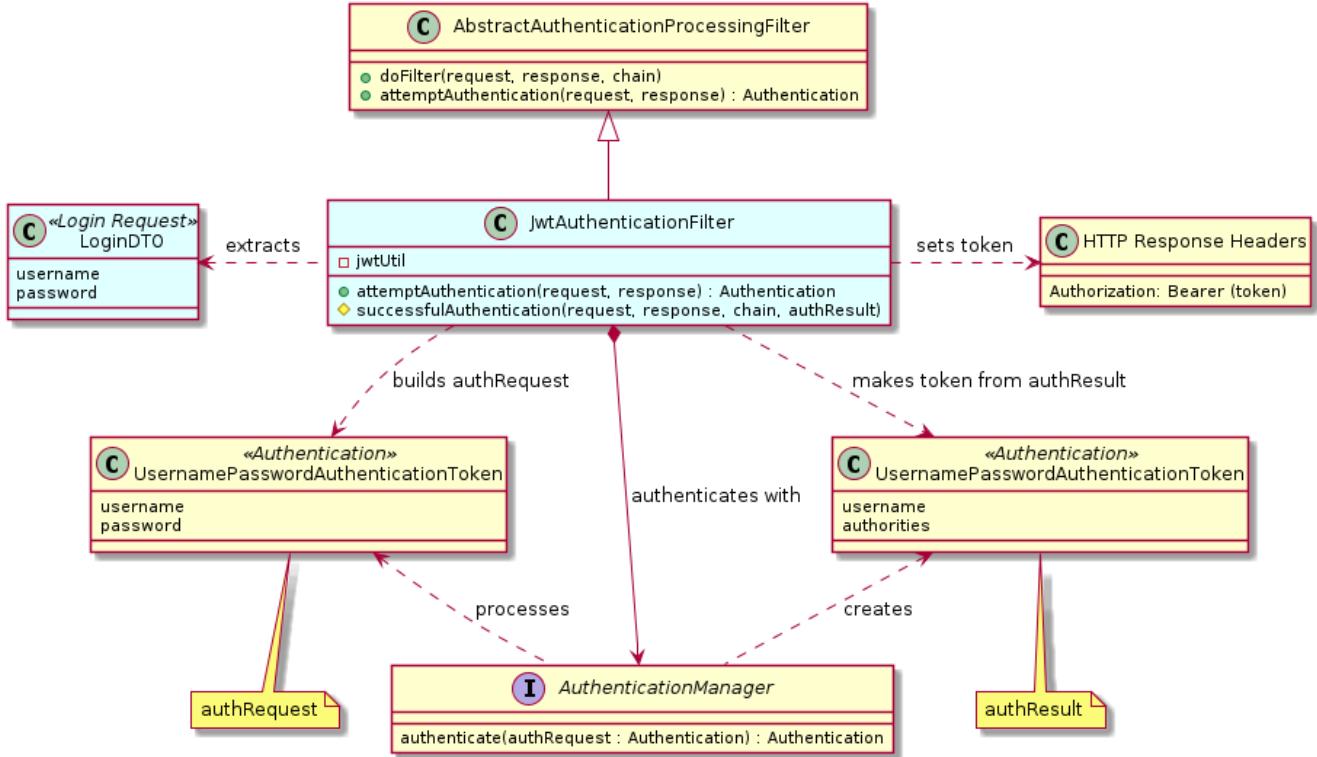


Figure 91. JwtAuthenticationFilter Relationships

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.

225.2. 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);
}
  
```

225.3. 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`

225.4. 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;
}
```

225.5. 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`

225.6. 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 226. 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
eyJhbGciOiJIUzIwMjQ4LC39.eyJzdWIiOiJmcmFzaWVyiwiWF0IjoxNTk0OTgwMTAyLCJleHAiOjE4OTQ5ODM3MDIsImF1dGgiOlSiUFJJQ0VfQ0hFQ0siLCJST0xFX0NVU1RPTUVSIi19.u2MmzTxaDoVNFGGCnraCWBuS_NS2NndZXkaT964hLgcDTvCYAW_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.

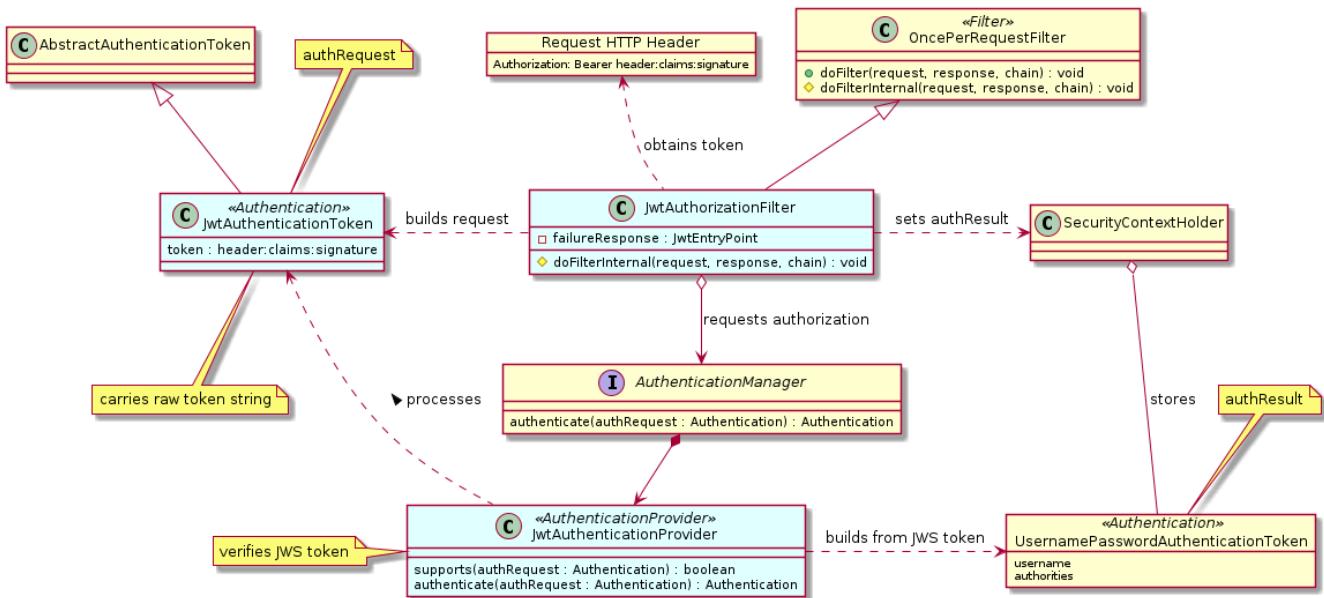
This class also relies on `JwtUtil` to implement the details of working with the JWS bearer token

JwtAuthorizationFilter

```
public class JwtAuthorizationFilter extends OncePerRequestFilter {  
    private final JwtUtil jwtUtil;  
    private final AuthenticationManager authenticationManager;  
    private final AuthenticationEntryPoint failureResponse = new JwtEntryPoint();
```

226.1. 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.

226.2. 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;
}
  
```

226.3. 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

226.4. 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.

226.5. 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 227. 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)

227.1. 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

```
@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.

227.2. 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

227.3. 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 228. 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

228.1. 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;
}
```

228.2. 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

228.3. 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  
eyJhbGciOiJIUzIwMjQ4NCiJ9.eyJzdWIiOiJzYW0iLCJpYXQiOjE1OTUwMTcwNDQsImV4cCI6MTg5NTAyMDY0NCwiY  
XV0aCI6WyJST0xFETUlOIl19.ICzAn1r2UyrgGJQSYk9uqxMAAq9QC1Dw7GKe0NiGvCyTasMfWSStrqxV6U  
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
eyJhbGciOiJIUzIwMjQNCJ9.eyJzdWIiOiJ3b29keSIsImhdCI6MTU5NTAxNzA1MSwiZXhwIjoxODk1MDIwNjUxL
CJhdXRoIjpbIlJPTEVfQ0xFUksiXX0.kreSFPgTIR2heGMLcjHFrqlydvhPZKR7Iy4F6b76WNIvAkbZVhfymbQ
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
eyJhbGciOiJIUzIwMjQNCJ9.eyJzdWIiOiJu3JtIiwiaWF0IjoxNTk1MDE3MDY1LCJleHAiOjE4OTUwMjA2NjUsI
mF1dGgiOlslsIuk9MRV9DVVNUT01FUijdfQ.UX4yPDu0LzWdEAObbjli0tZ7ePU1RSIH_o_hayPrlmNxhjU5DL6X
Q42iRCLLuFgw

$ 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
eyJhbGciOiJIUzIwMjQNCJ9.eyJzdWIiOiJmcmFzaWVyIiwiaWF0IjoxNTk1MDE3MDcxLCJleHAiOjE4OTUwMjA2N
zEsImF1dGgiOlslsIUFJJQ0VfQ0hFQ0siLCJST0xFX0NVU1RPTUVSIl19.ELAe5foIL_u2QyhpjwDoqQbL4Hl1Ik
uir9CJPdOT80w2lI5Z1GQY6ZaKvW883txI
```

① norm and frasier have role **CUSTOMER**

228.4. `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.eyJzdWIiOiJ3b29keSIsImhdCI6MTU5NTAxNzA1MSwiZhwIjoxODk1MDIwNjUxLCJhdXRoIjpbIlJPTEVfQ0xFUksiXX0.kreSFPgTlr2heGMLcjHFrglydvhPZKR7Iy4F6b76WNIvAkbZVhfymbQxekuPL-Ai" #woody  
[woody, [ROLE_CLERK]]  
  
$ curl -X POST http://localhost:8080/api/carts -H "Authorization: Bearer eyJhbGciOiJIUzM4NCJ9.eyJzdWIiOiJ3b29keSIsImhdCI6MTU5NTAxNzA1MSwiZhwIjoxODk1MDIwNjUxLCJhdXRoIjpbIlJPTEVfQ0xFUksiXX0.kreSFPgTlr2heGMLcjHFrglydvhPZKR7Iy4F6b76WNIvAkbZVhfymbQxekuPL-Ai" #woody  
{"url": "http://localhost:8080/api/carts", "message": "Forbidden", "description": "caller[woody] is forbidden from making this request", "date": "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 eyJhbGciOiJIUzM4NCJ9.eyJzdWIiOiJu3JtIiwiaWF0IjoxNTk1MDE3MDY1LCJleHAiOjE40TUwMjA2NjUsImF1dGgiOlslUk9MRV9DVVNUT01FUijdfQ.UX4yPDu0LzWdEAObbjli0tZ7ePU1RSIH_o_hayPrlmNxhjU5DL6XQ42iRCLLuFgw" #norm  
[norm, [ROLE_CUSTOMER]]  
  
$ curl -X POST http://localhost:8080/api/carts -H "Authorization: Bearer eyJhbGciOiJIUzM4NCJ9.eyJzdWIiOiJu3JtIiwiaWF0IjoxNTk1MDE3MDY1LCJleHAiOjE40TUwMjA2NjUsImF1dGgiOlslUk9MRV9DVVNUT01FUijdfQ.UX4yPDu0LzWdEAObbjli0tZ7ePU1RSIH_o_hayPrlmNxhjU5DL6XQ42iRCLLuFgw" #norm  
{"username": "norm", "items": []}
```

228.5. 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 eyJhbGciOiJIUzM4NCJ9.eyJzdWIiOiJmcmFzaWVyiwiWF0IjoxNTk1MDE3MDcxLCJleHAiOjE4OTUwMjA2NzEsImF1dGgiOlSiUFJJQ0VfQ0hFQ0siLCJST0xFX0NVU1RPTUVSIl19.ELAe5foIL_u2QyhpjwDoqQbL4Hl1Ikuir9CJPdOT80w2lI5Z1GQY6ZaKvW883txI" #frasier  
[frasier, [PRICE_CHECK, ROLE_CUSTOMER]]  
  
$ curl -X POST "http://localhost:8080/api/carts/items?username=norm&name=chardonnay"  
-H "Authorization: Bearer eyJhbGciOiJIUzM4NCJ9.eyJzdWIiOiJmcmFzaWVyiwiWF0IjoxNTk1MDE3MDcxLCJleHAiOjE4OTUwMjA2NzEsImF1dGgiOlSiUFJJQ0VfQ0hFQ0siLCJST0xFX0NVU1RPTUVSIl19.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", "date": "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 eyJhbGciOiJIUzM4NCJ9.eyJzdWIiOiJuub3JtIiwiaWF0IjoxNTk1MDE3MDY1LCJleHAiOjE4OTUwMjA2NjUsImF1dGgiOlSiUk9MRV9DVVNUT01FUijdfQ.UX4yPDu0LzWdEA0bbJli0tZ7ePU1RSIH_o_hayPrlmNxhjU5DL6XQ42iRCLLuFgw" #norm  
{"username": "norm", "items": ["beer"]}
```

228.6. 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 eyJhbGciOiJIUzIwMjQ4MDIyfQ.eyJzdWIiOiJmcmFzaWVyaWFiZjoxNTk1MDE3MDcxLCJleHAiOjE4OTUwMjA2NzEsImF1dGgiOlslsiUFJJQ0VfQ0hFQ0siLCJST0xFX0NVU1RPTUVSIi19.ELAe5foIL_u2QyhpjwDoqQbL4Hl1Ikuir9CJPdOT80w2lI5Z1GQY6ZaKvW883txI" #frasier
{"url":"http://localhost:8080/api/carts?username=norm","message":"Forbidden","description":"caller[frasier] is forbidden from making this request","date":"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 eyJhbGciOiJIUzIwMjQ4MDIyfQ.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 eyJhbGciOiJIUzIwMjQ4MDIyfQ.eyJzdWIiOiJ3b29keSIsImhdCI6MTU5NTAxNzA1MSwiZXhwIjoxODk1MDIwNjUxLCJhdXRoIjpbIlJPTEVfQ0xFUksixX0.kreSFPgTIR2heGMLcjHFrglydvhPZKR7Iy4F6b76WNIvAkbZVhfymQxekuPL-Ai" #woody
{"username":"norm","items":["beer"]}
```

228.7. 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 eyJhbGciOiJIUzIwMjQ4NCJ9.eyJzdWIiOiJ3b29keSIsImhdCI6MTU5NTAxNzA1MSwiXhwIjoxODk1MDIwNjUxLCJhdXRoIjpbIlJPTEVfQ0xFUksixX0.kreSFPgTlr2heGMLcjHFrglydvhPZKR7Iy4F6b76WNIvAkbZVhfymbQxekuPL-Ai" #woody
{"url":"http://localhost:8080/api/carts?username=norm","message":"Forbidden","description":"caller[woody] is forbidden from making this request","date":"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 not longer there.

Sam Can Remove Norms Cart

```
$ curl -X GET http://localhost:8080/api/whoAmI -H "Authorization: Bearer eyJhbGciOiJIUzIwMjQ4NCJ9.eyJzdWIiOiJzYW0iLCJpYXQiOjE1OTUwMTcwNDQsImV4cCI6MTg5NTAyMDY0NCwiYXV0aCI6WyJST0xFX0FETUlOI19.ICzAn1r2UyrgPJQSYk9uqxMAAq9QC1Dw7GKe0NiGvCyTasMfWSStrqxV6Uiit-cb4" #sam
[sam, [ROLE_ADMIN]]
```



```
$ curl -X DELETE http://localhost:8080/api/carts?username=norm -H "Authorization: Bearer eyJhbGciOiJIUzIwMjQ4NCJ9.eyJzdWIiOiJzYW0iLCJpYXQiOjE1OTUwMTcwNDQsImV4cCI6MTg5NTAyMDY0NCwiYXV0aCI6WyJST0xFX0FETUlOI19.ICzAn1r2UyrgPJQSYk9uqxMAAq9QC1Dw7GKe0NiGvCyTasMfWSStrqxV6Uiit-cb4" #sam
```



```
$ curl -X GET http://localhost:8080/api/carts -H "Authorization: Bearer eyJhbGciOiJIUzIwMjQ4NCJ9.eyJzdWIiOiJub3JtIiwiaWF0IjoxNTk1MDE3MDY1LCJleHAiOjE4OTUwMjA2NjUsImF1dGgiOlsiUk9MRV9DVVNUT01FUijdfQ.UX4yPDu0LzWdEA0bbJli0tZ7ePU1RSIH_o_hayPrlmNxhjU5DL6XQ42iRCLLuFgw" #norm
{"url":"http://localhost:8080/api/carts","message":"Not Found","description":"no cart found for norm","date":"2020-07-17T20:50:59.465210Z"}
```

Chapter 229. 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**

Enabling HTTPS

copyright © 2021 jim stafford (jim.stafford@jhu.edu)

Chapter 230. 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.

230.1. 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

230.2. 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
5. implement a Maven Failsafe integration test using HTTPS

Chapter 231. 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 slightly encoded (`dXNlcjpwYXNzd29yZA==`), plaintext password was issued in the clear. We can fix that by enabling HTTPS.

Chapter 232. 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. [\[45\]](#) [\[46\]](#)

232.1. 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. [\[47\]](#)

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.

232.2. 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. [\[48\]](#) 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 later is necessary for clients. The truststore can be shared but its contents need to be trusted.

232.3. 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.

232.4. Self Signed Certificates

The words "trust" and "verify" were used a lot in the paragraphs above when describing certificates.

When we visit various web sites—that locked icon next to the "https" URL indicates the certificate presented by the server was verified and came from a trusted source.

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).

[45] "[HTTPS](#)", Wikipedia

[46] "[Transport Layer Security](#)", Wikipedia

[47] "[Public key certificate](#)", Wikipedia

[48] "[Spring Boot HTTPS](#)", ZetCode, July 2020

Chapter 233. 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.

233.1. 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. I kept the values of the certificate extremely basic since there is little chance we will ever use this in a trusted environment.

Generate Self-signed RSA Certificate

```
$ keytool -genkeypair -keyalg RSA -keysize 2048 -validity 3650 \①
-keystore keystore.p12 -alias https-hello \②
-storepass password
What is your first and last name?
[Unknown]: localhost
What is the name of your organizational unit?
[Unknown]:
What is the name of your organization?
[Unknown]:
What is the name of your City or Locality?
[Unknown]:
What is the name of your State or Province?
[Unknown]:
What is the two-letter country code for this unit?
[Unknown]:
Is CN=localhost, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=Unknown correct?
[no]: yes
```

① specifying a valid date 10 years in the future

② assigning the alias `https-hello` to what is generated in the keystore

233.2. 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.

233.3. Add TLS properties

The following shows a minimal set of properties needed to enable TLS. [49]

Example TLS properties

```
server.port=8443 ①  
server.ssl.enabled=true  
server.ssl.key-store=classpath:keystore.p12 ②  
server.ssl.key-store-password=password ③  
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".

[49] "[HTTPS using Self-Signed Certificate in Spring Boot](#)", Baeldung, June 2020

Chapter 234. 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 \
-H "Authorization: BASIC dXNlcjpwYXNzd29yZA=="
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.

Chapter 235. Accept Self-signed Certificates

curl and older 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
```

235.1. Optional Redirect

To handle clients that may address our application using the wrong protocol or port number—we can optionally setup 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() {
    var 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() {
    var connector = new Connector("org.apache.coyote.http11.Http11NioProtocol");
    connector.setScheme("http");
    connector.setPort(8080);
    connector.setSecure(false);
    connector.setRedirectPort(8443);
    return connector;
}
```

235.2. 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

235.3. Follow Redirects

Browsers automatically follow redirects and we can get curl to automatically follow redirects by adding the `--location` option (or `-L` abbreviated). 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 -kvL -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==
>
< 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
*   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.
> GET /api/authn/hello?name=jim HTTP/1.1
> Host: localhost:8443
> Authorization: Basic dXNlcjpwYXNzd29yZA==
>
< HTTP/1.1 200
hello, jim
```

① `-L` (`--location`) redirect option causes curl to follow the 302/Redirect response

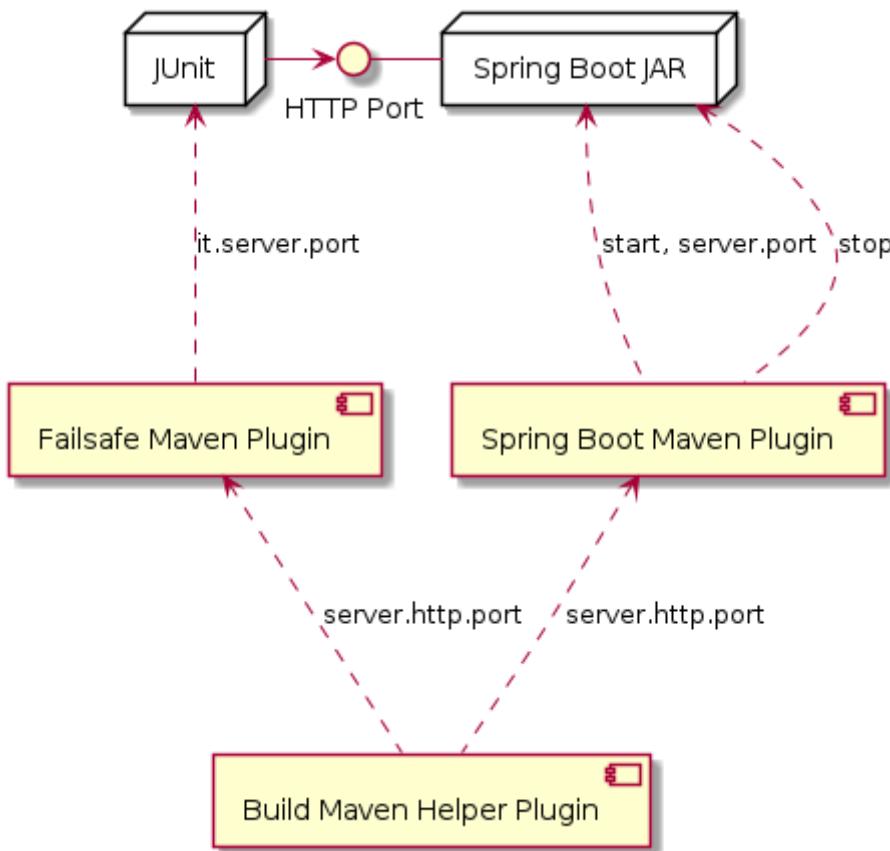
235.4. Caution About Redirects

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.

Eliminating the optional redirect also eliminates the need for the redirect code and reduces our required steps to obtaining the certificate and setting a few simple properties.

Chapter 236. Maven Integration Test

Since we are getting close to real deployments to test environments and we have hit unit integration tests pretty hard, I wanted to demonstrate a test of the HTTPS configuration using a true integration test and the Maven Failsafe plugin.



A Maven Failsafe 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. 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.

Figure 92. Maven Failsafe Integration Test

With that said, we will still want to be able to execute simple integration tests like this within the IDE. Therefore expect some setup aspects to support both IDE-based and Maven-based integration testing setup in the paragraphs that follow.

236.1. 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

We will make use of three (3) plugins to perform that work within Maven. Each is also accompanied by *steps to mimic the Maven capability on a small scale with the IDE*:

- [**spring-boot-maven-plugin**](#) - used to start and stop the server-side Spring Boot process

- (use IDE, "java -jar" command, or "mvn springboot:run" command to manually start, restart, and stop the server)
- **build-maven-helper-plugin** - used to allocate a random network port for server
 - (within the IDE you will use a property file that uses a well-known port# used one-at-a-time)
- **maven-failsafe-plugin** - used to run the JUnit JVM with the tests—passing in the port#—and verifying/asserting the results.
 - (use IDE to run test following server-startup)

236.2. Spring Boot Maven Plugin

The `spring-boot-maven-plugin` will be configured with at least 2 executions to support Maven integration testing.

spring-boot-maven-plugin Shell

```
<plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
  <executions>
    ...
  </executions>
</plugin>
```

236.2.1. SpringBoot: pre-integration-test Phase (start)

The following snippet shows the plugin being used 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 use a separate plugin to generate the port number and have that assigned to the Maven `server.http.port` property at build time. The client-side Spring Boot Test 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
- 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/S listen port

Failsafe is overriding the fixed value from `application-https.properties`.

`src/main/resources/application-https.properties`

```
server.port=8443
```

The above execution phase has the same impact as if we launched the JAR manually with `spring.profiles.active` and whether `server.port` was supplied on the command. 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/https-hello-example-*-.jar --spring.profiles.active=https
Tomcat started on port(s): 8443 (https) with context path '' ①

$ java -jar target/https-hello-example-*-.jar --spring.profiles.active=https
--server.port=7712 ②
Tomcat started on port(s): 7712 (http) with context path '' ②
```

- ① Spring Boot using well-known/fixed port# supplied in `application-https.properties`
- ② Spring Boot using runtime `server.port` property to override port to use

236.2.2. SpringBoot: post-integration-test Phase (stop)

The following snippet shows the Spring Boot plugin being used to `stop` a running server.

```
<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

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.



236.3. Build Helper Maven Plugin

The **build-helper-maven-plugin** contains **various utilities** that are helpful to create a repeatable, portable build. We are using the **reserve-network-port** goal to select an available HTTP port at build-time. The allocated port number is assigned to the Maven **server.http.port** property. This was shown picked up by the Spring Boot Maven Plugin earlier.

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>build-helper-maven-plugin</artifactId>
  <executions>
    <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>
  </executions>
</plugin>
```

- 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

236.4. Failsafe Plugin

The **Failsafe plugin** has some default behavior, but once we start configuring it — we need to restate much of what it would have done automatically for us.

maven-failsafe-plugin Declaration

```
<plugin>
  <artifactId>maven-failsafe-plugin</artifactId>
  <executions>
    ...
  </executions>
</plugin>
```

236.4.1. Failsafe: integration-test Phase

In the snippet below, we are primarily configuring Failsafe to launch the JUnit test with an `it.server.port` property. This will be read in by the `ServerConfig @ConfigurationProperties` class

integration-test Phase

```
<execution>
  <id>integration-test</id>
  <phase>integration-test</phase> ①
  <goals> ①
    <goal>integration-test</goal>
  </goals>
  <configuration>
    <includes> ①
      <include>**/*IT.java</include>
    </includes>
    <systemPropertyVariables> ②
      <it.server.port>${server.http.port}</it.server.port>
    </systemPropertyVariables>
    <additionalClasspathElements> ③
      <additionalClasspathElement>
        ${basedir}/target/classes</additionalClasspathElement>
      </additionalClasspathElements>
      <useModulePath>false</useModulePath> ④
    </configuration>
  </execution>
```

① re-state some Failsafe default relative to phase, goal, includes

② add a `-Dit.server.port=${server.http.port}` system property to the execution

③ adding `target/classes` to classpath when JUnit test using classes from "src/main"

④ turning off some Java 9 module features



Full disclosure. I need to refresh my memory on exactly why default `additionalClasspathElements` and `useModulePath` did not work here.

236.4.2. Failsafe: verify Phase

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 pass/fail assertion is delayed a few phases so that the build does not fail while integration resources are still running.

verify Phase

```
<execution>
  <id>verify</id>
  <phase>verify</phase>
  <goals>
    <goal>verify</goal>
  </goals>
</execution>
```

236.5. JUnit @SpringBootTest

With the Maven setup complete—that brings us back to a familiar looking JUnit test and `@SpringBootTest`. However, there is no application or server-side resources in the Spring context,

```
@SpringBootTest(classes={ClientTestConfiguration.class}, ①
                 webEnvironment = SpringBootTest.WebEnvironment.NONE) ②
@ActiveProfiles({"its"}) ③
public class HttpsRestTemplateIT {
  @Autowired ④
  private RestTemplate authnUser;
  @Autowired ⑤
  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

③ activate `its` profile for scope of test case

④ inject `RestTemplate` configured with user credentials that can authenticate

⑤ inject URL to endpoint test will be calling



Since we have no `RANDOM_PORT` and a late `@LocalPort` injection, we can move `ServerConfig` to the configuration class and inject the baseURL product.

236.6. ClientTestConfiguration

This trimmed down `@Configuration` class is all that is needed for JUnit test to be a client of a remote process.

```

@Configuration(proxyBeanMethods = false)
@EnableConfigurationProperties //used to set it.server properties
@EnableAutoConfiguration      //needed to setup logging
@Slf4j
public class ClientTestConfiguration {

    ...
    @Bean
    @ConfigurationProperties("it.server")
    public ServerConfig itServerConfig() { ... }

    @Bean
    public URI authnUrl(ServerConfig serverConfig) { ...① }

    @Bean
    public RestTemplate authnUser(RestTemplateBuilder builder,...②
    ...
}

```

① baseUrl of remote server

② RestTemplate with authentication and HTTPS filters applied

236.7. application-its.properties

The following snippet shows the `its` profile-specific configuration file, complete with

- `it.server.scheme` (`https`)
- `it.server.port` (`8443`)
- trustStore properties pointing at the server-side identity keystore.

application-its.properties

```

it.server.scheme=https
#must match self-signed values in application-https.properties
it.server.trust-store=keystore.p12
it.server.trust-store-password=password
#used in IDE, overridden from command line during failsafe tests
it.server.port=8443 ①

```

① default port when working in IDE. Overridden by command line properties by Failsafe



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.

236.8. username/password Credentials

The following shows the username and password credentials being injected using values from the properties. 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;
```

236.9. 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`.

```
@Bean  
@ConfigurationProperties("it.server")  
public ServerConfig itServerConfig() {  
    return new ServerConfig();  
}
```

From the property file earlier, you will notice that the URL scheme will be "https" and the port will be "8443" or whatever property override is supplied on the command line. The resulting value will be injected into the `@Configuration` class.

236.10. authnUrl URI

Since we don't have the late-injected `@LocalPort` for the web-server and our `ServerConfig` is now all property-based, we can now delegate baseUrls to injectable beans. The following shows the `baseUrl` from `ServerConfig` being used to construct a URL for "api/authn/hello".

Building baseUrl from Injected ServerConfig

```
@Bean  
public URI authnUrl(ServerConfig serverConfig) {  
    URI baseUrl = serverConfig.getBaseUrl();  
    return UriComponentsBuilder.fromUri(baseUrl).path("/api/authn/hello").build()  
.toUri();  
}
```

236.11. authUser RestTemplate

By no surprise, `authnUser()` is adding a `BasicAuthenticationInterceptor` containing the injected username and password to a new `RestTemplate` for use in the test. It also has `ServerConfig` injected and uses that to test for the use of HTTPS. If HTTPS is being used, an HTTPS-based client request factory is added.

authnUser RestTemplate

```
@Bean
public RestTemplate authnUser(RestTemplateBuilder builder,
                               ServerConfig serverConfig,
                               ClientHttpRequestFactory requestFactory) {
    RestTemplate restTemplate = builder.requestFactory(
        //used to read the streams twice -- so we can use the logging filter below
        ()->new BufferingClientHttpRequestFactory(new SimpleClientHttpRequestFactory(
    )))
        .interceptors(new BasicAuthenticationInterceptor(username, password),
                     new RestTemplateLoggingFilter())
        .build();
    if (serverConfig.isHttps()) {
        log.info("enabling SSL requests");
        restTemplate.setRequestFactory(requestFactory);
    }
    return restTemplate;
}
```

236.12. HTTPS ClientHttpRequestFactory

The HTTPS-based ClientHttpRequestFactory is built by following some [excellent instructions](#) and short article provided by Geoff Bourne. The following intermediate factory relies on the ability to construct an SSLContext.

```
import org.springframework.http.client.HttpComponentsClientHttpRequestFactory;
import org.springframework.http.client.ClientHttpRequestFactory;
/*
TLS configuration based on great/short article by Geoff Bourne
https://medium.com/@itzgeoff/using-a-custom-trust-store-with-resttemplate-in-spring-boot-77b18f6a5c39
*/
@Bean
public ClientHttpRequestFactory httpsRequestFactory(SSLContext sslContext) {
    HttpClient httpsClient = HttpClientBuilder.create()
        .setSSLContext(sslContext)
        .build();
    return new HttpComponentsClientHttpRequestFactory(httpsClient);
}
```

236.13. SSL Context

The SSLContext [@Bean](#) factory locates and loads the trustStore based on the properties within [ServerConfig](#). If found, it uses the [SSLContextBuilder](#) from the apache HTTP libraries to create a [SSLContext](#).

```

import org.apache.http.ssl.SSLContextBuilder;
import javax.net.ssl.SSLContext;
...
@Bean
public SSLContext sslContext(ServerConfig serverConfig) {
    URL trustStoreUrl = HttpsExampleApp.class.getResource "/" + serverConfig
.getTrustStore();
    if (null==trustStoreUrl) {
        throw new IllegalStateException("unable to locate truststore:/"
+ serverConfig.getTrustStore());
    }
    try {
        return SSLContextBuilder.create()
            .loadTrustMaterial(trustStoreUrl, serverConfig.getTrustStorePassword(
))
            .setProtocol("TLSv1.2")
            .build();
    } catch (Exception ex) {
        throw new IllegalStateException("unable to establish SSL context", ex);
    }
}

```

236.14. JUnit @Test

The core parts of the JUnit test are pretty basic once we have the HTTPS/Authn-enabled `RestTemplate` and `baseUrl` injected. From here it is just a normal test, but activity is remote on the server side.

```

public class HttpsRestTemplateIT {
    @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

236.15. Maven Verify

When we execute `mvn verify` (with option to add `clean`), we see the port being determined and assigned to the `server.http.port` Maven property.

Starting Maven Build

```

$ mvn verify
...
- build-helper-maven-plugin:3.1.0:reserve-network-port (reserve-network-port)
Reserved port 52024 for server.http.port ①
...②
- maven-surefire-plugin:3.0.0-M5:test (default-test) @ https-hello-example ---
...
- spring-boot-maven-plugin:2.4.2:repackage (package) @ https-hello-example ---
Replacing main artifact with repackaged archive
③
- spring-boot-maven-plugin:2.4.2:start (pre-integration-test) @ https-hello-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

236.15.1. Server Output

When the server starts, we can see that the `https` profile is active and Tomcat was assigned the `52024` port value from the build.

Server Output

```
HttpsExampleApp#logStartupProfileInfo:664 The following profiles are active: https ①
TomcatWebServer#initialize:108 Tomcat initialized with port(s): 52024 (https) ②
TomcatWebServer#start:220 Tomcat started on port(s): 52024 (https) with context path
'' ③
```

① `https` profile has been activated on the server

② server HTTP(S) port assigned to `52024`

236.15.2. JUnit Client Output

When the JUnit client starts, we can see that SSL is enabled and the baseURL contains `https` and the dynamically assigned port `52024`.

JUnit Client Output

```
HttpsRestTemplateIT#logStartupProfileInfo:664 The following profiles are active: its
①
ClientTestConfiguration#authnUrl:64 baseUrl=https://localhost:52024 ②
ClientTestConfiguration#authnUser:107 enabling SSL requests ③
```

① `its` profile is active in JUnit client

② `baseUrl` is assigned `https` and port `52024`, with the latter dynamically assigned at build-time

③ SSL has been enabled on client

236.15.3. 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, application/*+json, text/xml, application/*+xml, */",
authorization:"Basic dXNlcjpwYXNzd29yZA==", host:"localhost:52024", connection:"Keep-
Alive", user-agent:"masked", accept-encoding:"gzip,deflate"]]
```

236.15.4. Failsafe Test Results

Test results are reported.

Failsafe Test Results

```
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 2.409 s - in  
info.ejava.examples.svc.https.HttpsRestTemplateIT  
[INFO] Results:  
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

236.15.5. Server is Stopped

Server is stopped.

Server is Stopped

```
[INFO] --- spring-boot-maven-plugin:2.4.2:stop (post-integration-test)  
[INFO] Stopping application...  
15:29:42.178 RMI TCP Connection(4)-127.0.0.1 INFO  
XBeanRegistrar$SpringApplicationAdmin#shutdown:159 Application shutdown requested.
```

236.15.6. Test Results Asserted

Test results are asserted.

Overall Test Results

```
[INFO]  
[INFO] --- maven-failsafe-plugin:3.0.0-M5:verify (verify) @ https-hello-example ---  
[INFO] -----  
[INFO] BUILD SUCCESS
```

Chapter 237. 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
- how to setup and run a Maven Failsafe Integration Test

Assignment 3: Security

copyright © 2021 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 238. Assignment Starter

There is a project within `race-starters/assignment3-race-security/secure-race-app` that contains some ground work for the security portions of the assignment (AOP coming). It contains:

1. a set of `@Configuration` classes within the `SecurityConfiguration` class. Each `@Configuration` class or construct is profile-constrained to match a section of the assignment.
2. unit integration tests that align with the sections of the assignment. Each test case activates one or more profiles identified by the assignment.
3. the shell of a few server-side implementation classes that are not profile-specific

The meat of the assignment is focused in the following areas

1. configuring a `WebSecurityConfigurerAdapter` to meet the different security levels of the assignment. Each profile will start over. You may end up copy/pasting between the different configurations between profiles.
2. the unit tests are well populated and anticipated that you will leverage them at some level.
 - a. you may use them to address the testing portion of the assignment. Each of the tests have been `@Disabled` and had the Racer Registration references commented out—waiting for your attention. If you enable the Race and Racer tests, they can help you identify configuration errors still left in your `WebSecurityConfigurerAdapter` changes.
 - b. you may simply reference them and write your own tests. They do demonstrate a few new JUnit concepts of `@TestInstance(PER_CLASS)` (to be able to make use of Spring context bean within a `@ParameterizedTest`) and `@Nested` tests (to organize tests)—so its worth referencing for a few tricks.
3. For the other server-side portions of the assignment
 - a. there is a skeletal `SecurityController` that lays out portions of the `whoAmI` and `authorities` endpoints.
 - b. there is a skeletal `SecureRacerRegistrationServiceImpl` and `@Configuration` class that could be used to optionally wrap your assignment 2 implementation (replace `Object` with your interfaces). The satisfactory alternative is to move your assignment 2 implementations into the assignment 3 module tree. Neither path (layer versus copy/edit) will get you more or less credit. Choose the path that makes the most sense to you.



If you layer your assignment3 over your assignment2 solution as separate modules, you will need to make sure that your assignment2 JARs are stored in the repository as vanilla JARs and not Spring Boot executable JARs (i.e., turn off the `spring-boot:repackage` goal).

Chapter 239. Assignment Support

The assignment support module(s) in `race-support/race-support-security` again provide some examples and ground work for you to complete the assignment—by adding a dependency. I chose to use a layered approach to security for Races and Racers. 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 `race-support-security-svc` adds this layer to the `race-support-api-svc` components.

The following dependency can replace your current dependency on the `race-support-api-svc`.

race-support-security-svc Dependency

```
<dependency>
    <groupId>info.ejava.assignments.security.race</groupId>
    <artifactId>race-support-security-svc</artifactId>
    <version>${ejava.version}</version>
</dependency>
```

The most notable class in the "support" module(s)—which should save you some work—is the `AuthorizationTestHelperConfiguration` class. The later JUnit tests in the "starter" module shows when this `@Configuration` class can be added to the test configuration to provide many injectable user/role-specific RestTemplates.

Chapter 240. Assignment 3a: Security Authentication

240.1. Anonymous Access

240.1.1. 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

240.1.2. 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.

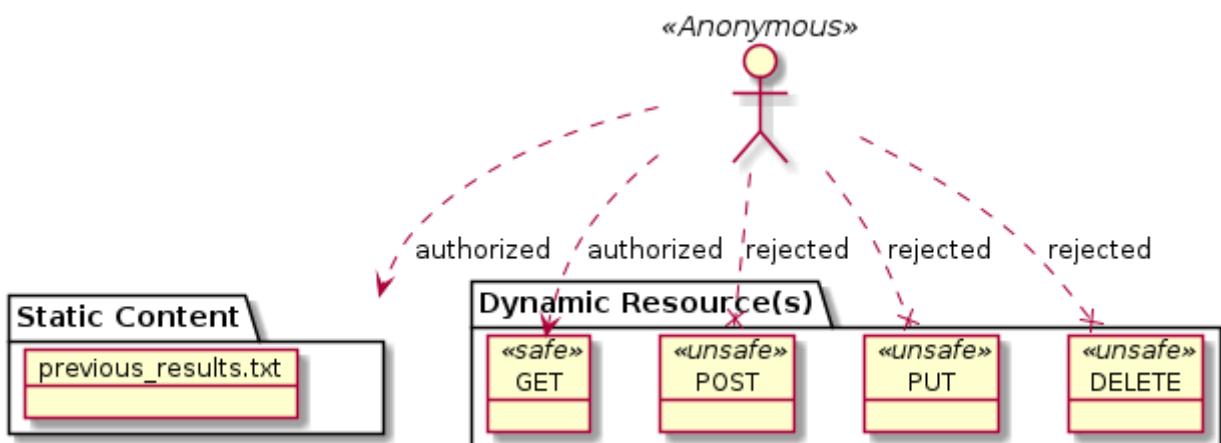


Figure 93. Anonymous Access

240.1.3. Requirements

1. Add a static text file `past_results.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_results.txt`

```
$ curl -X GET http://localhost:8080/content/past_results.txt
Past Race Results
```

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/`

2. Configure anonymous access for the following resources methods

- static content below `/content`



Configure WebSecurity to ignore all calls below `/content/`. Leverage the `antMatcher()` to express the `pattern`.

- safe calls (`GET`) for races, racers, and racer registration resources



Configure HttpSecurity to permit all GET calls matching URIs below races, racers, and racer registrations. Leverage the `antMatcher()` to express the method and `pattern`.

3. Configure authenticated access for the following resource operations

- non-safe calls (`POST`, `PUT`, and `DELETE`) for races, racers, and racer registration resources



Configure HttpSecurity to authenticate any request that was not yet explicitly permitted

4. Create a unit integration test case that verifies

- anonymous user access granted to static content
- anonymous user access granted to a safe (`GET`) resource call to each resource type (races, racers, and racer registrations)
- anonymous user access denial to a non-safe resource call to each resource type.



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.

5. Restrict this security configuration to the `authenticated-access` profile and activate that profile while testing this section.

```
@Configuration  
@Profile("anonymous-access") ①  
public class PartA1_AnonymousAccess extends  
WebSecurityConfigurerAdapter {
```

① restrict configuration changes to the `anonymous-access` profile



```
@SpringBootTest(classes= {...},  
@ActiveProfiles({"test", "anonymous-access"}) ①  
@DisplayName("Part A1: Anonymous Access")  
public class AnonymousAccessNTest {
```

① activate the `anonymous-access` profile (with any other designed profiles) when executing tests

240.1.4. 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

240.1.5. Additional Details

1. No accounts are necessary for this portion of the assignment. All testing is performed using an anonymous caller.

240.2. Authenticated Access

240.2.1. 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 `WebClient` client
2. locate the current authenticated user identity

240.2.2. Overview

In this portion of the assignment you will be authenticating with the API (using `RestTemplate` or `WebClient`) and tracking the caller's identity within the application.

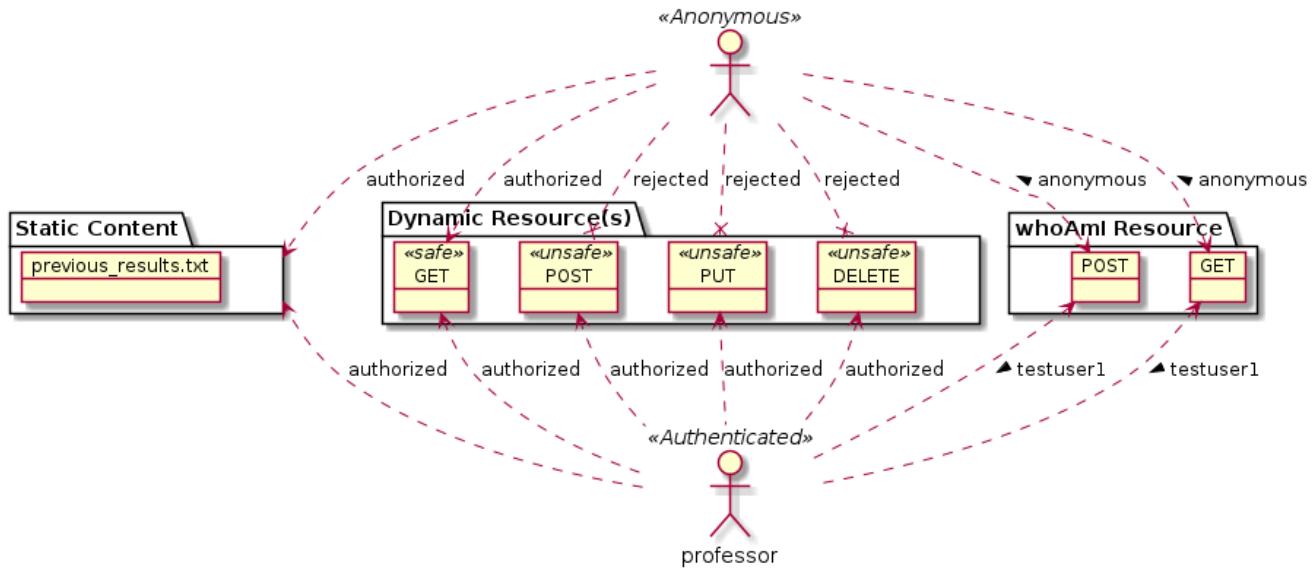


Figure 94. Authenticated Access

You're starting point should be an application with functional Races, Racers, and RacerRegistrations API services. Races and Racers were provided to you. You implemented RacerRegistration in assignment2. All functional tests for RaceRegistration are passing.

240.2.3. Requirements

1. Configure the application to use a test username/password of `professor/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`

a. supply two access methods: **GET** and **POST**. Configure security such that neither require authentication.



Configure HttpSecurity to permit all method requests below `/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 + null + close_paren)



You may inject or lookup the user details for the caller identity within the server-side method.

6. Create a set of unit integration tests that demonstrate the following using `RestTemplate` or `WebClient`:

- 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. authenticated access to a one POST/create race, racer, and racer registration resource operation
d. successful identification of the authenticated caller identity using the `whoAmI` resource operations

7. Restrict this security configuration to the `authenticated-access` profile and activate that profile during testing this section.



```
@Configuration
@Profile("authenticated-access") ①
public class AuthenticationPart extends WebSecurityConfigurerAdapter {
    ====
    @SpringBootTest(classes={...},
        webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
    @ActiveProfiles({"test", "authenticated-access"}) ②
    public class AuthenticatedNTest {
```

① activated with `authenticated-access` profile

② activating desired profiles

8. Establish a security configuration that is active for the `noauth` profile which allows for but requires no credentials to call each resource/method. This will be helpful for simplify some demonstration scenarios where security is not essential.



```
@Configuration
@Profile("noauth")
public class AllowAll extends WebSecurityConfigurerAdapter {
```

The `noauth` profile will allow unauthenticated access to operations that are also free of CSRF checks.



```
curl -X POST http://localhost:8080/api/races -H 'Content-Type: application/json' -d '{"name":"race"}'  
{"id": "2", "name": "race", "eventLength": 0.0, "eventDate": null, "location": null}
```

240.2.4. Grading

Your solution will be evaluated on:

1. add an authenticated identity to `RestTemplate` or `WebClient` 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 method for the race, racer, and racer registration 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).

240.2.5. Additional Details

1. Not a requirement, but strive to have the different types of client access injected into the test case so that the individual test cases and tests do not repeat boilerplate setup.

```
@TestConfiguration  
public class RaceTestConfiguration {  
    @Value("${spring.security.user.name:}")  
    private String username;  
    @Value("${spring.security.user.password:}")  
    private String password;  
    @Bean  
    @ConditionalOnProperty(prefix = "spring.security.user", name={"name"})  
    public String authnUsername() { return username; }  
    @Bean  
    public RestTemplate anonymousUser(RestTemplateBuilder builder) {...}  
    @Bean  
    @ConditionalOnProperty(prefix = "spring.security.user",  
                           name = {"name", "password"})  
    public RestTemplate authnUser(RestTemplateBuilder builder) {...}}
```

240.3. User Details

240.3.1. Purpose

In this portion of the assignment, you will demonstrate your knowledge of assembling a `UserDetailsService` to track the credentials of all users in 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

240.3.2. 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.

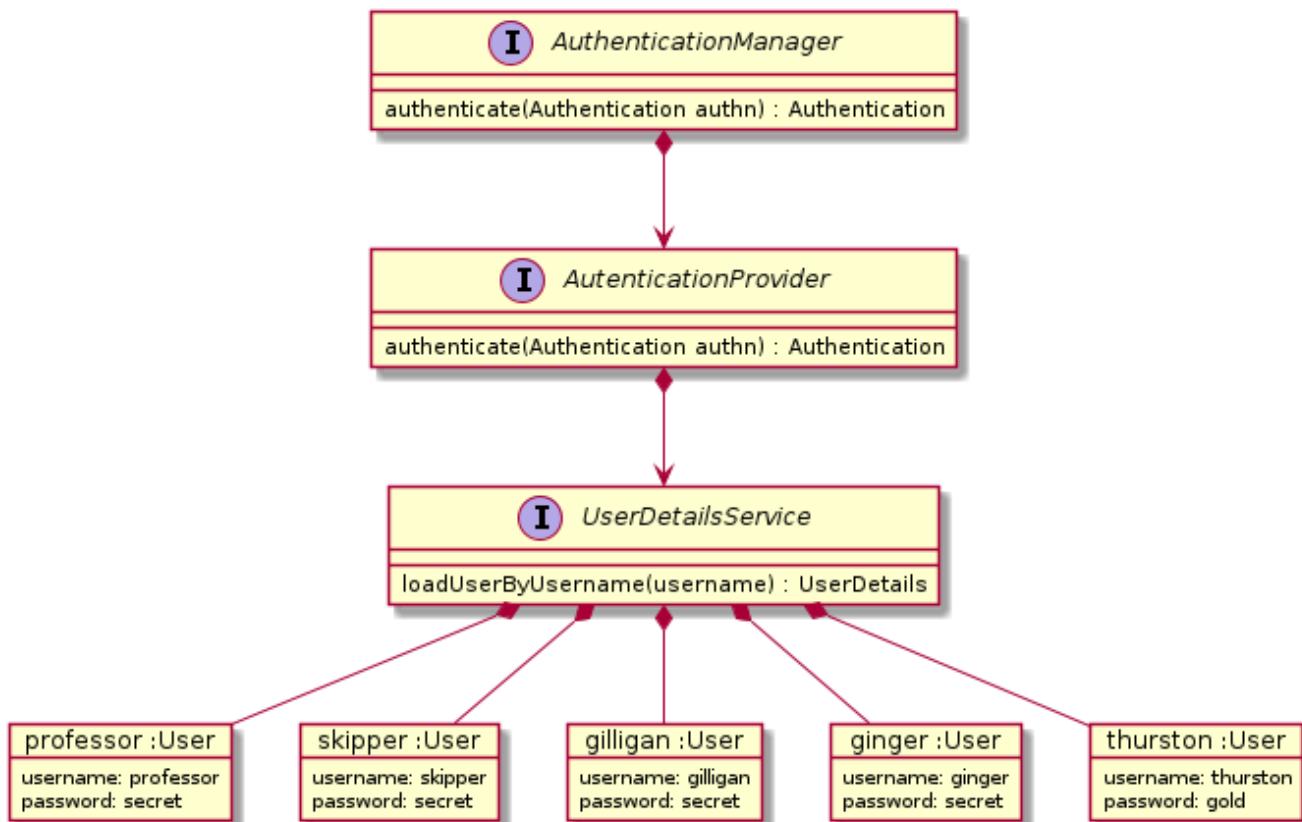


Figure 95. User Details

The `races-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 `RaceAccounts @ConfigurationProperty` class.

1. professor/secret
2. skipper/secret
3. gilligan/secret
4. ginger/secret
5. thurston/gold

240.3.3. 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 `WebSecurityConfigurers`.



```
@Bean  
public UserDetailsService userDetailsService(PasswordEncoder  
encoder) {
```

- b. populate it with the 5 users from an injected `RaceAccounts @ConfigurationProperty` bean
- 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:

- a. activates the `userdetails` profile
- b. verifies successful authentication and identification of the authenticated username using the `whoAmI` resource
- c. verifies successful access to a one POST/create race, racer, and racer registration resource operation for each user

240.3.4. 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 race, racer, and racer registration
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

240.3.5. 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, `@TestInstance(TestInstance.Lifecycle.PER_CLASS)` must be used in order to reference Spring context resources from the method source.
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. This can occur when independently testing `createRacer` and `createRacerRegistration` because later rules will limit a caller to a single racer.

Chapter 241. Assignment 3b: Security Authorization

241.1. Authorities

241.1.1. Purpose

In this portion of the assignment, you will demonstrate your knowledge of authorities. You will:

1. define role-based and permission-based authorities

241.1.2. 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 96. Authorities Test Resource

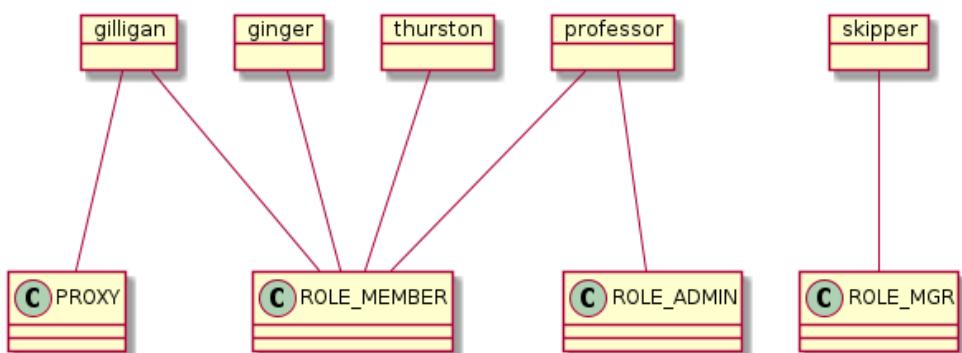


Figure 97. Assignment Principles and Authorities

The `races-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. professor: ROLE_ADMIN, ROLE_MEMBER
2. skipper: ROLE_MGR (no role member)
3. gilligan: ROLE_MEMBER, PROXY
4. ginger: ROLE_MEMBER
5. thurston: ROLE_MEMBER

241.1.3. Requirements

1. Create an `api/authorities` resource with a `GET` method
 - a. accepts an "authority" query parameter
 - b. returns (textual) "TRUE" or "FALSE" depending on whether the caller has the authority

- assigned
2. Create one or more unit integration test cases that
 - a. activates the `authorities` profile
 - b. verifies the unauthenticated caller has no authorities assigned
 - c. verifies each the authenticated callers have proper assigned authorities

241.1.4. 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

241.1.5. 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(TestInstance.Lifecycle.PER_CLASS)` in order to leverage the Spring context in the `@MethodSource`.

241.2. Authorization

241.2.1. 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

241.2.2. Overview

In this portion of the assignment you will be restricting access to specific resource operations based on path and expression-based resource restrictions.

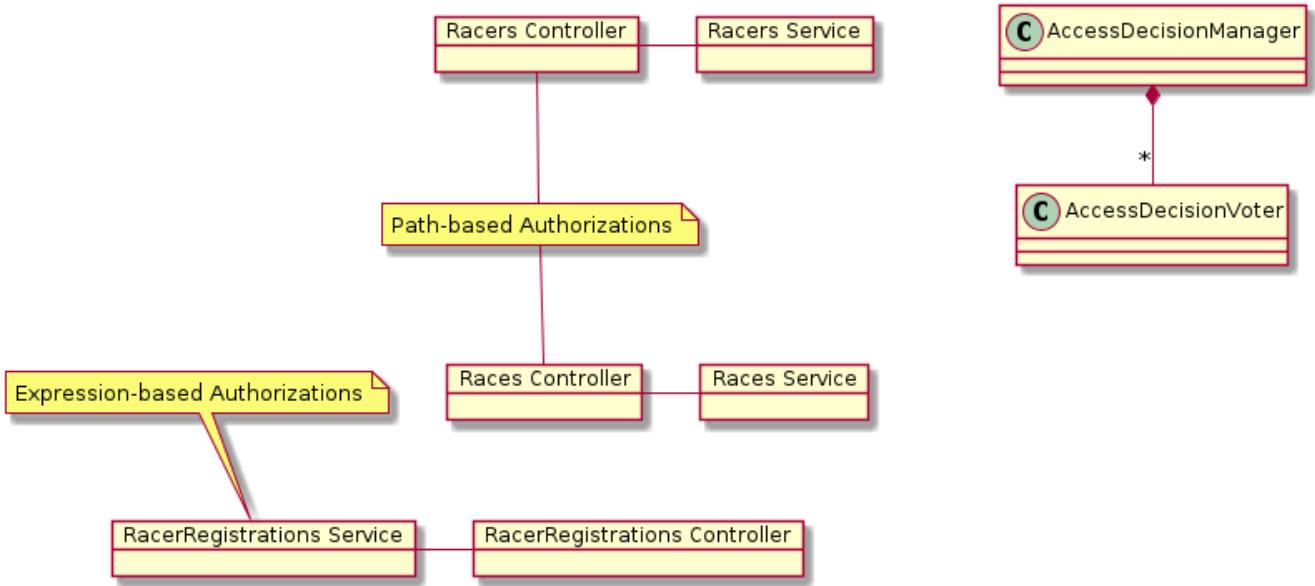


Figure 98. Authorization

241.2.3. Requirements

1. Update the resources to store the identity of the caller with the resource created to identify the owner
 - a. race will store the username (as ownername) of the race creator (provided)
 - b. racer will store the username (as username) of the racer it represents (provided)
 - c. Racer Registration should store the username (as username) of the racer it represents (new)
 - d. the identity should not be included in the marshalled DTO returned to callers
2. Define access constraints for resources using path and expression-based authorizations
 - a. Use path-based authorizations for race and racer resources, assigned to the URIs since you are not working with modifiable source code for these two resources



Configure HttpSecurity to require callers of Races and Racers to have specific roles

- b. Use expression-based authorizations for racer registration resources, applied to the service methods



Use annotations on Racer Registration service methods to trigger authorization checks. Remember to enable global method security for the `prePostEnabled` annotation form for your application and tests.

3. Restrict access according to the following

- a. Races (path-based authorizations)
 - i. continue to restrict non-safe methods to authenticated users (from Authenticated Access)
 - ii. authenticated users may modify races that match their login (provided)
 - iii. authenticated users may delete races that match their login or have the MGR role (provided)

- iv. only authenticated users with the **ADMIN** role can delete all races (new)
 - b. Racers (path-based authorizations)
 - i. *continue to restrict non-safe methods to authenticated users (from Authenticated Access)*
 - ii. *authenticated users may create a single resource to represent themselves (provided)*
 - iii. *authenticated users may modify a racer resource that matches their login (provided)*
 - iv. *authenticated users may delete a racer resource that matches their login or have the **MGR** role (provided)*
 - v. only authenticated users with the **ADMIN** role can delete all racers (new)
 - c. RacerRegistrations (annotation-based authorizations)
 - i. authenticated users may create a registration for only themselves and an existing race (new)
 - ii. authenticated users with the **MGR** role or **PROXY** permission may register racers for a race - to be owned by those users (i.e., caller "skipper" or "gilligan" can register existing racer "ginger" for a race) (new)
 - iii. authenticated users may only delete a registration for a racer matching their username (new)
 - iv. authenticated users may delete any registration if they have the **MGR** role (new)
 - v. authenticated users with the **ADMIN** role can delete all race registrations (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

241.2.4. 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 race resource URIs
2. implement annotation-based authorization constraints
 - a. whether expression-based authorization constraints were properly defined and user for racer and race registration 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

241.2.5. Additional Details

1. Role inheritance can be defined using a **RoleHierarchy** bean.

241.3. HTTPS

241.3.1. 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 a Maven Failsafe integration test using HTTPS

241.3.2. Overview

In this portion of the assignment you will be configuring your server to support HTTPS only when the **https** profile is active.

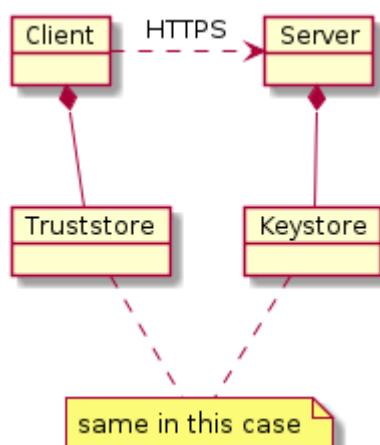


Figure 99. Authorization

241.3.3. 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

2. Implement an IT/Failsafe integration test that will
 - a. start the server with `https` profile active
 - b. configure and start JUnit with an integration test
 - c. establish an HTTPS connection with `RestTemplate` or `WebClient`
 - d. successfully invoke using HTTPS and evaluate the result

241.3.4. 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

241.3.5. 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. You might try implementing the integration test with HTTP before switching to HTTPS to eliminate too many concurrent changes.

Chapter 242. Assignment 3c: 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 behavior of components. As a specific example, you are going to correct a problem I saw with Races and Racers at the end of assignment 2 without changing the Races/Racers source code. Races and Racers allow the caller to provide an `id` and should not have. A null `id` value would have resulted in a one-up `id` value generated and managed on the server-side.

You are going to implement a set of logic that will reject the creation of a Race and Racer using a specific `id`.

The first two sections (reflection and dynamic proxies) of the AOP assignment lead up to the final solution (aspects) in the third section.

No Races/Racers Compilation Dependencies



The `src/main` portions of the assignment must have no compilation dependency on Races and Racers. All compilation dependencies and most knowledge of Races and Racers will be in the JUnit tests.

242.1. Reflection

242.1.1. Purpose

In this portion of the assignment, you will demonstrate your knowledge of using reflection to obtain and invoke a method proxy. You will:

1. obtain a method reference and invoke it using Java Reflection

242.1.2. Overview

In this portion of the assignment you will implement a set of helper methods for a base class (`NullPropertyAssertion`) located in the security "support" module you have been using—tasked with validating whether objects validated have nulls for identified fields. If the specified property exists and is not null—an exception will be thrown by the base class. The derived class—which will be completed by you—will assist in locating the method reference to the "getter" and invoking it to obtain the current property value.

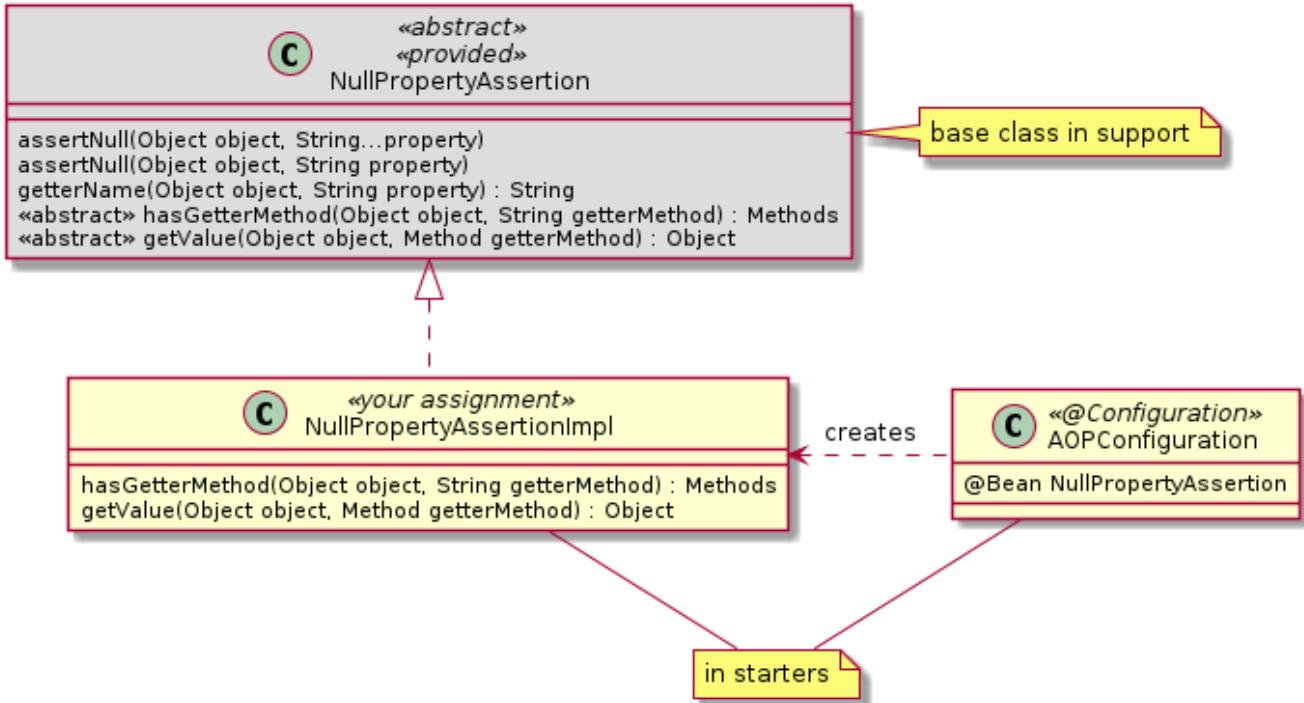


Figure 100. AOP and Method Proxies

The base class has been added to the `races-security-svc` "support" module. You will find an implementation class shell, `@Configuration` class with `@Bean` factory, and JUnit test in the assignment 3 security "starter".

No RaceDTO/RacerDTO Compilation Dependency

i Note that you see no mention of `RaceDTO` or `RacerDTO` in the above description/diagram. Everything will be accomplished using Java reflection.

242.1.3. Requirements

1. implement the `NullPropertyAssertionImpl#hasGetMethod` to locate and return the `java.lang.reflect.Method` for the requested (getter) method name.
 - a. return the method object if it exists
 - b. return null if it does not exist — this is not an error if it does not exist



It is not considered an error for this class if the `getPropertyName` requested does not exist for the target class. JUnit tests—which know the full context of the calls—will decide if the result is correct or not.

2. implement the `NullPropertyAssertionImpl#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
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-RaceDTO/RacerDTO objects on purpose. The validator under test must work with any type of object passed in. Only "getter" access will be supported for this capability. No "field" access will be performed.

242.1.4. 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

242.1.5. Additional Details

1. The JUnit test is supplied and should not need to be modified beyond re-enabling it.

242.2. Dynamic Proxies

242.2.1. 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

242.2.2. 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 non-null check of specific properties of objects passed as parameters to the target object.

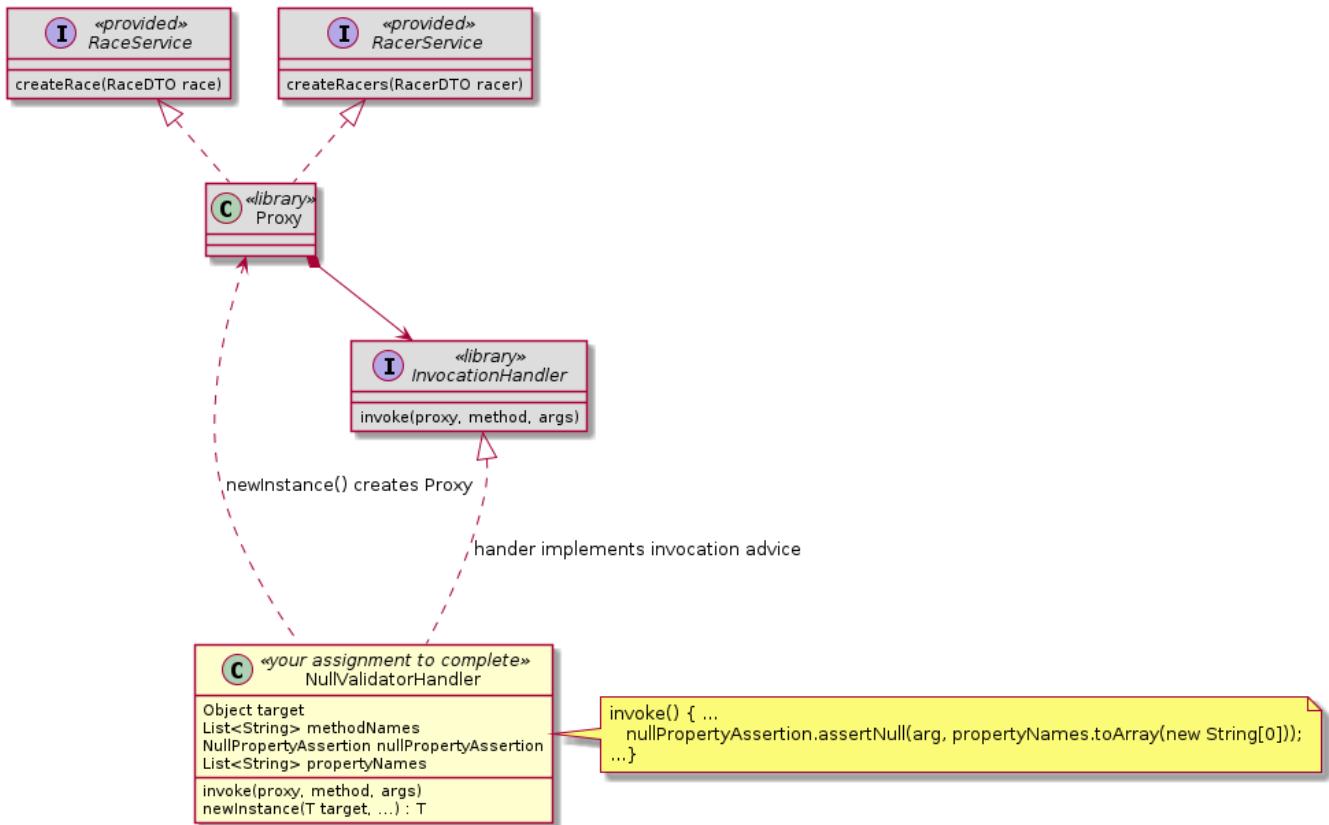


Figure 101. Dynamic Proxy Advice

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 `RacesService` and `RacersService` from the Spring context.



No RaceDTO/RacerDTO Compilation Dependency

There will be no mention of or direct dependency on Races or Racers. Everything will be accomplished using Java reflection.

242.2.3. Requirements

1. implement the details for the `NullValidatorHandler` class that
 - a. implements the `java.lang.reflect.InvocationHandler` interface
 - b. has implementation properties
 - i. target object that it is the proxy for
 - ii. (literal) `methodNames` it will operate against on the target object
 - iii. `nullPropertyAssertion` instance (from previous section)
 - iv. `propertyNames` it will test (used also in 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



`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 methods matching the list of `methodNames`. Those arguments will have properties checked that match the list of `propertyNameNames`.
 - 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
2. Use the provided JUnit tests in the "starter" module to verify the functionality of the requirements above.

242.2.4. 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

242.2.5. Additional Details

1. The JUnit test case uses real RacesService and RacerServices @Autowired from the Spring context, but only instantiates the proxy as a POJO within the test case.
2. 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 following Aspect solution—since Spring interface Aspects are implemented using Dynamic Proxies.

242.3. Aspects

242.3.1. 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

242.3.2. Overview

In this portion of the assignment you will implement a `ValidatorAspect` that will "advise" `createRace` and `createRacer` calls to provided Races/Racers services.

The aspect will be part of your overall Spring context and will be able to change the behavior of the Races and Racers used within your runtime application and tests when activated. The aspect will specifically reject any object passed to these two methods that have a non-null `id` property. The aspect will be defined to match the targeted methods and arguments but will have no compilation dependency that is specific to Races or Racers.

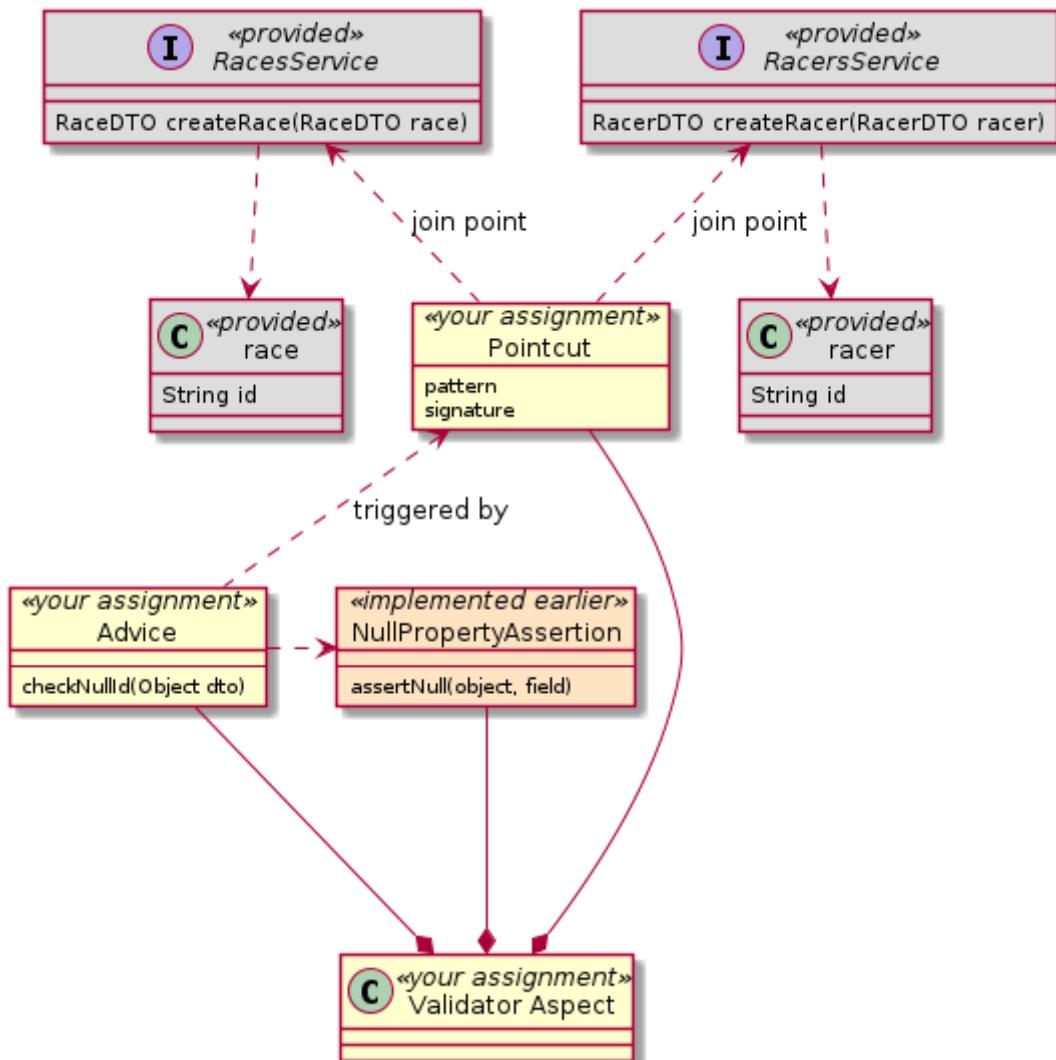


Figure 102. Aspect Advice



No RaceDTO/RacerDTO Compilation Dependency

There will be no direct dependency on Races or Racers. Everything will be accomplished using AOP expressions and Java reflection constructs.

242.3.3. Requirements

1. add AOP dependencies using the `spring-boot-starter-aop` module
2. enable AspectJ auto proxy handling within your application

3. create a **ValidatorAspect** component
 - a. inject a **NullPropertyAssertion** bean
 - b. make it an aspect
 - c. make it conditional on the **aop** profile



This means the Race and Racer services will act as delivered when the profile is not active and reject non-null ID fields when the profile is active.

4. define a "pointcut" that will target the create methods of the RacesService and RacersService. This pointcut should both:
 - a. define a match pattern for the "join point"
 - b. identify an argument signature available for the "advice"
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
 - c. invokes the **NullPropertyAssertion** bean with the **dto** parameter passed to the create method and asks to validate that the **id** field is null
6. Use the provided JUnit test cases to verify completion of the above requirements
 - a. the initial **NoAspectSvcNTest** deactivates the **aop** profile to demonstrate baseline behavior we want to change/not-change
 - b. the second **AspectSvcNTest** activates the **aop** profile and asserts different results
 - c. the third **AspectWebNTest** is there for added demonstration that the aspect was added to what was injected into the Races and Racers controllers.

242.3.4. 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

242.3.5. Additional Details

1. The JUnit test case uses real RacesService and RacerServices @Autowired from the Spring context augmented with aspects. Some of those aspects are security-related. Your aspect will be included when the `aop` profile is active.
2. Ungraded activity—create a breakpoint in the Advice and Races/RacersService(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 © 2021 jim stafford (jim.stafford@jhu.edu)

Chapter 243. Introduction

Many times, business logic must execute additional behavior that is outside of 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)

243.1. 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

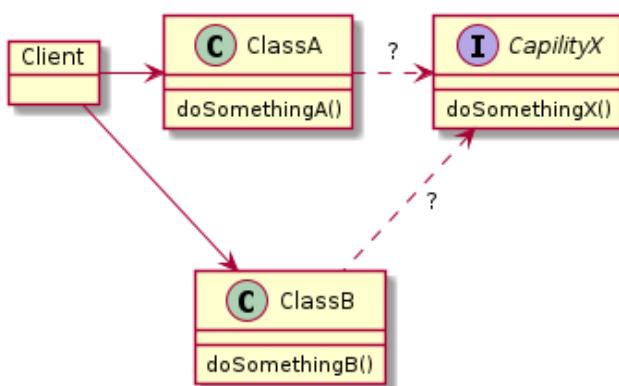
243.2. Objectives

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. implement dynamically assigned behavior to methods using Spring Aspect-Oriented Programming (AOP) and AspectJ
5. identify method join points to inject using pointcut expressions
6. implement advice that executes before, after, and around join points
7. implement parameter injection into advice

Chapter 244. Rationale

Our problem starts off with two independent classes depicted as `ClassA` and `ClassB` and a caller labelled 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 may get you in trouble.

Figure 103. New Cross-Cutting Design Decision

244.1. 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?

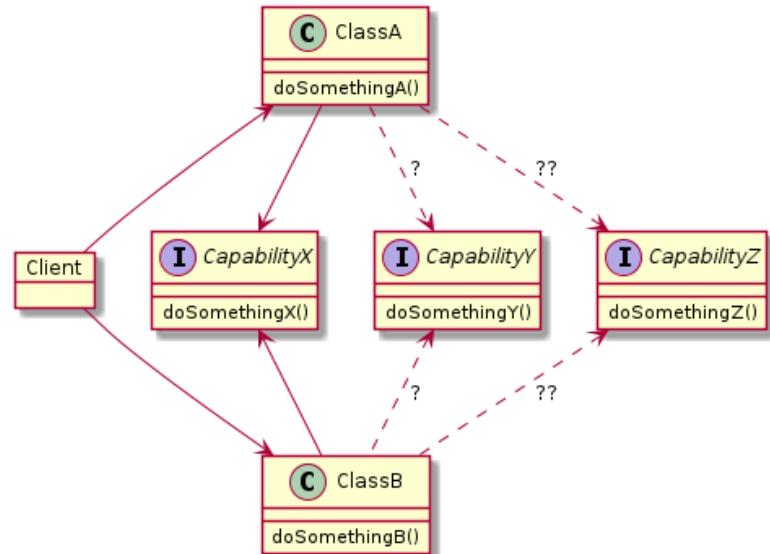
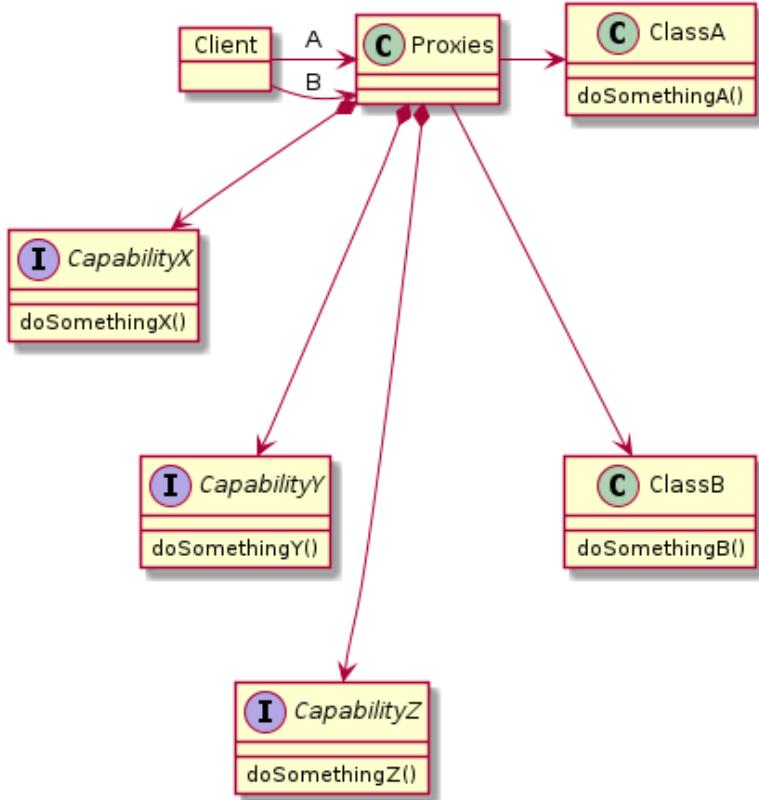


Figure 104. More Cross-Cutting Capabilities

244.2. Using Proxies



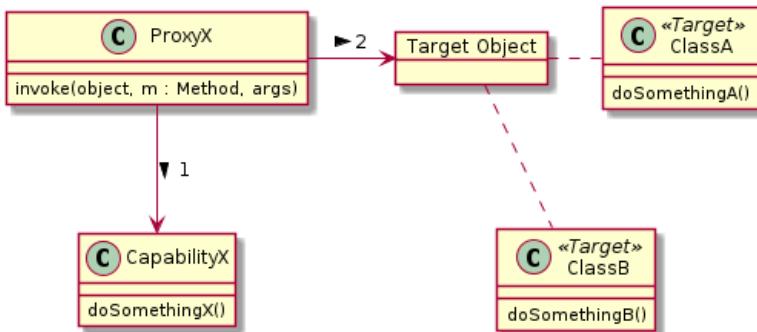
What we can do instead is leave `ClassA` and `ClassB` alone and wrap calls to them in a series of one or more proxied calls. `X` might perform auditing, `Y` might perform normalization of inputs, and `Z` might make sure the connection to the database is available and a transaction active. All we need `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. Lets begin to solve this with Java Reflection.

Chapter 245. 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.



Lets 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.

245.1. Reflection Method

Java Reflection provides the means to obtain a handle to **Fields** and **Methods** of a class. In the example below, I show code that obtains a reference to the `createItem` method, in the **ItemsService** interface, and accepting objects 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") provided 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)
```

245.2. 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 = ...  
  
//invoke method using target object and args  
Object[] args = new Object[] { BedDTO.bedBuilder().name("Bunk Bed").build() }; ②  
log.info("invoke calling: {}({})", method.getName(), args);  
  
Object result = method.invoke(bedsService, args); ③  
  
log.info("invoke {} returned: {}", method.getName(), result);
```

① we must obtain a target object to invoke

② arguments are passed into `invoke()` using a varargs array

③ 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))
```

245.3. 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 246. 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. [50]

246.1. 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, and 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

246.2. 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 `ItemsService` interface supplied at runtime

② `Serializable` interface implemented by DynamicProxy implementation class

246.3. 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

246.4. 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

246.5. `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. You should recall the `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.

246.6. 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.

[50] "Dynamic Proxy Classes", Oracle JavaSE 8 Technotes

Chapter 247. CGLIB

Code Generation Library (CGLIB) is a byte instrumentation library that allows the manipulation or creation of classes at runtime. ^[51]

Where JDK Dynamic Proxies implement a proxy behind an interface, CGLIB dynamically implements a sub-class 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).

247.1. 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 sub-class 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

247.2. 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

247.3. 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

247.4. 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))
```

[51] "Introduction to cglib", Baeldung, Aug 2019

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 can we 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 same class, calling method of manually instantiated class, and calling method of injected object.

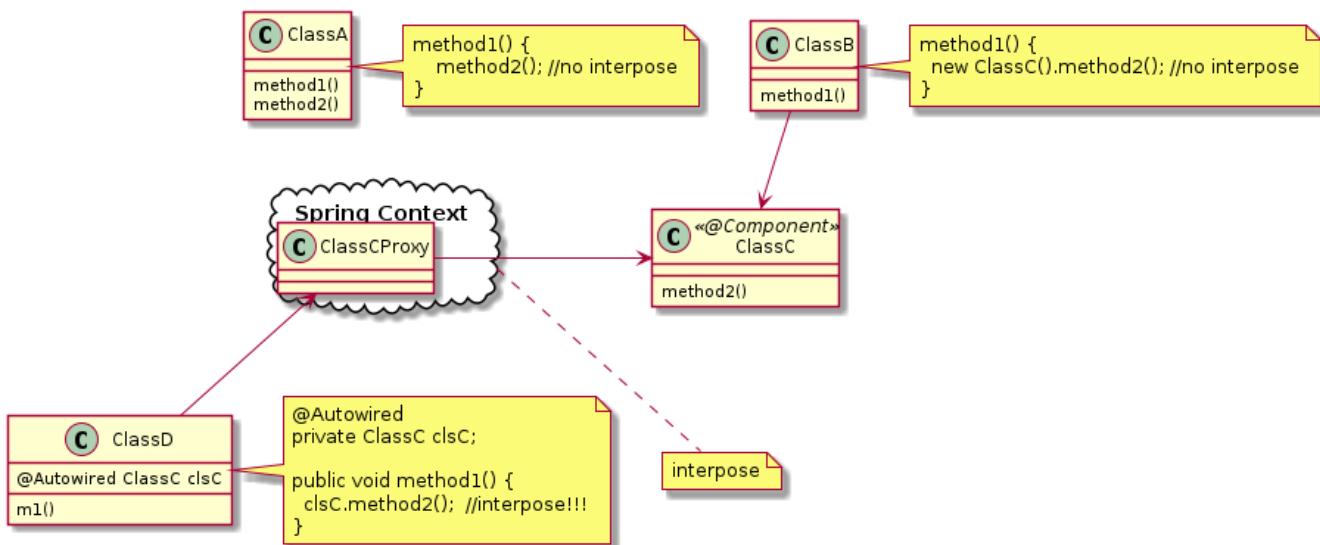


Figure 105. Interpose is only Automatic for Injected Components

- Buddy Method:** For the `ClassA` with `m1()` and `m2()` in the same class, Spring will make no attempt to interpose a proxy in between those two methods. 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()`.
- 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 that is used to locate the join point and how that is all modularized through a concept called aspect.

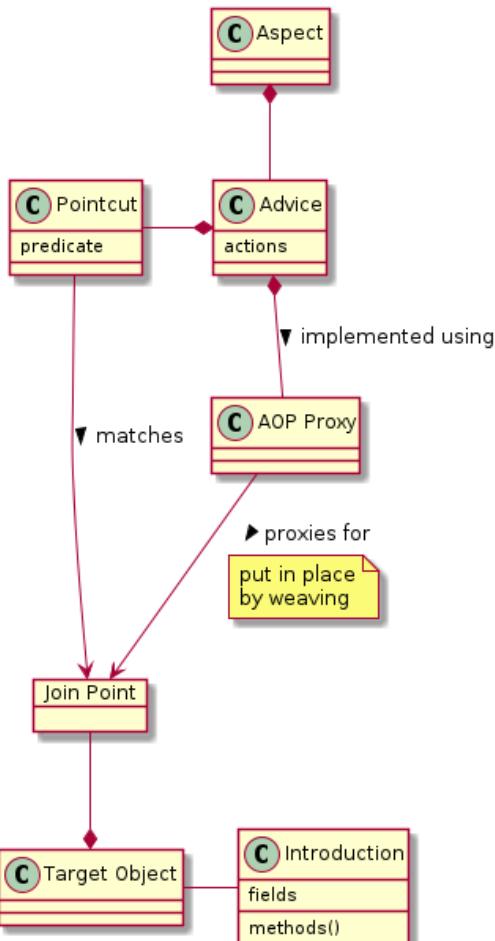


Figure 106. AOP Key Terms

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 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
public class ...
```

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
- signature with name and parameters

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 signature defines a name and parameter types for the pointcut expression

249.5. Pointcut Expression

The [Spring AOP](#) pointcut expressions use the 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
• target	match the target for a given type — useful when injecting typed advice arguments
• @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 (**this**, **target**, and **@annotation**) 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 services 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
```

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

@Before	runs prior to calling join point
@AfterReturning	runs after successful return from join point
@AfterThrowing	runs after exception from join point
@After	runs after join point no matter — i.e., finally
@Around	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, lets 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. ^[52]

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 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` when used in the following way. The difference will show up when we later use them to inject typed arguments into the advice. These are considered "contextual" designators and are primarily placed in the predicate to pull out members of the call for injection.

```
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

```
@target(org.springframework.stereotype.Service) ①
@annotation(org.springframework.core.annotation.Order) ②
```

- ① matches all methods in class annotated with `@Service`
- ② matches all methods having annotation `@Order`

[52] "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 `JoinPoint` 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 is 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.

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<BedsDTO>` interface and the `proxy` name is mapped to a vanilla `java.lang.Object`. The `target` type mapping constrains this to calls to the `BedsServiceImpl`.

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 (i.e., no proxy layer) 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 were 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`—in order to delegate to the target and handle the result. Lets take a look at each in order to have a complete set of examples.

To demonstrate, I am going to define an 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 calls:

1. the first will be successful
2. the second will throw a `NotFound RuntimeException`.

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) {
    log.info("afterThrowingMowerUpdate: {}, {} => {}", id, mowerUpdate, ex.toString());
}
```

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.

@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))
```

252.5. @Around

@Around is the most capable advice but possibly the more expensive one 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());
        result = ex;
        throw ex;
    } finally {
        log.info("returning after aroundMowerUpdate: {}, {} => {}", id,
mowerUpdate, (result==null ? null :result.toString()));
    }
}
```

The **@Around** advice example will log activity prior to calling the join point, after successful return from join point, and finally after all advice complete.

@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 prior to calling the join point, after an exception from the join point, and finally after all advice complete.

@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. Other Features

We have covered a lot of capability in this chapter and likely all you will need. However, know there were a few other topics left unaddressed that I thought might be of interest in certain circumstances.

- [Ordering](#) - useful when we declare multiple advice for the same join point and need one to run before the other
- [Introductions](#) - a way to add additional state and behavior to a join point/target instance
- [Programmatic Spring AOP proxy creation](#) - a way to create Spring AOP proxies on the fly versus relying on injection. This is useful for data value objects that are typically manually created to represent a certain piece of information.
- [Schema Based AOP Definitions](#) - Spring also offers an means to express AOP behavior using XML. They are very close in capability—so if you need the ability to flexibly edit aspects in production without changing the Java code—this is an attractive option.

Chapter 254. 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 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

After learning this material you will surely be able to automatically envision the implementation techniques used by Spring in order 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.

Heroku Deployments

copyright © 2021 jim stafford (jim.stafford@jhu.edu)

Chapter 255. Introduction

To date we have been worrying about the internals of our applications, how to configure them, test them, interface with them, and how to secure them. We need others to begin seeing our progress as we continue to fill in the details to make our applications useful.

In this lecture—we will address deployment to a cloud provider. We will take a hands-on look at deploying to Heroku—a cloud platform provider that makes deploying Spring Boot and Docker-based applications part of their key business model without getting into more complex hosting frameworks.

255.1. Goals

You will learn:

- to deploy an application under development to a cloud provider to make it accessible to Internet users
- to deploy incremental and iterative application changes

255.2. Objectives

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

1. create a new Heroku application with a choice of names
2. deploy a Spring Boot application to Heroku using the Heroku Maven Plugin
3. interact with your developed application on the Internet
4. make incremental and iterative changes

Chapter 256. Heroku Background

According to their website, [Heroku](#) is a cloud provider that provides the ability to "build, deliver, monitor, and scale apps". They provide a fast way to go from "idea to URL" by bypassing company managed infrastructure. ^[53]

There are many cloud providers but not many are in our sweet spot of offering a platform for Spring Boot and Docker applications without the complexity of bare metal OS or a Kubernetes cluster. They also offer these basic deployments for [no initial cost](#) for non-commercial applications—such as proof of concepts and personal projects that stay within a 512MB memory limit.

There is a lot to Heroku that will not be covered here. However, this lecture will provide a good covering of how to achieve successful deployment of a Spring Boot application. In a follow-on lecture we will come back to Heroku to deploy Docker images and see the advantages it doing so. The following lists a few resources on the Heroku web site

- [Working with Spring Boot](#)
- [Developing with Docker](#)

[53] "[What is Heroku](#)", Heroku Web Site, July 2020

Chapter 257. Setup Heroku

You will need to setup an account with Heroku in order to use their cloud deployment environment. This is a free account and stays free until we purposely choose otherwise. If we exceed free constraints — our deployment simply will not run. There will be no surprise bill.

- visit the [Heroku Web Site](#)
- select [Sign Up For Free]
- create a free account and complete the activation
 - I would suggest skipping 2-factor authentication for simplicity for class use. You can always activate it later.
 - Salesforce bought Heroku and now has some extra terms to agree to
- install the [command line interface \(CLI\)](#) for your platform. It will be necessary to work at the shell level quite a bit
 - refer to the [Heroku CLI reference as necessary](#)

Chapter 258. Heroku Login

Once we have an account and the CLI installed—we need to login using the CLI. This will redirect us to the browser where we can complete the login.

Heroku Command Line Login

```
$ heroku login  
heroku: Press any key to open up the browser to login or q to exit:  
Opening browser to https://cli-auth.heroku.com/auth/cli/browser/f944d777-93c7-40af-  
b772-0a1c5629c609  
Logging in... done  
Logged in as ...
```

Chapter 259. Create Heroku App

At this point you are ready to perform a one-time (per deployment app) process that will reserve an app-name for you on herokuapp.com. When working with Heroku—think of app-name as a deployment target with an Internet-accessible URL that shares the same name. For example, my app-name of `ejava-boot` is accessible using <https://ejava-boot.herokuapp.com>. I can deploy one of many Spring Boot applications to that app-name (one at a time). I can also deploy the same Spring Boot application to multiple Heroku app-names (e.g., integration and production)

Let jim have ejava :)



Please use your own naming constructs. I am kind of fond of the `ejava-` naming prefix.

Example Create Heroku App

```
$ heroku create [app-name] ①  
Creating ⚡ [app-name]... done  
https://app-name.herokuapp.com/ | https://git.heroku.com/app-name.git
```

① if app-name not supplied, a random app-name will be generated



Heroku also uses Git repositories for deployment

Heroku creates a Git repository for the app-name that can also be leveraged as a deployment interface. I will not be covering that option.

You can create more than one heroku app and the app can be renamed with the following `apps:rename` command.

Example Rename Heroku App

```
$ heroku apps:rename --app oldname newname
```

Visit the [Heroku apps](#) page to locate technical details related to your apps.

Heroku will try to determine the resources required for the application when it is deployed the first time. Sometimes we have to give it details (e.g., provision DB)

Chapter 260. Create Spring Boot Application

For this demonstration, I have created a simple Spring Boot web application (docker-hello-example) that will be part of a series of lectures this topic area. Don't worry about the "Docker" naming for now. We will be limiting the discussion relative to this application to only the Spring Boot portions during this lecture.

260.1. Example Source Tree

The following structure shows the simplicity of the web application.

Example Spring Boot Web Application Source Tree

```
docker-hello-example/
|-- pom.xml
`-- src/main/java/info/ejava/examples/svc/docker
    |-- hello
    |   |-- DockerHelloExampleApp.java
    |   '-- controllers
    |       |-- ExceptionAdvice.java
    |       '-- HelloController.java
    '-- resources
        '-- application.properties
```

260.1.1. HelloController

The supplied controller is a familiar "hello" example, with optional authentication. The GET method will return a hello to the name supplied in the `name` query parameter. If authenticated, the controller will also issue the caller's associated username.

HelloController

```
@RestController
public class HelloController {
    @GetMapping(path="/api/hello",
                 produces = {MediaType.TEXT_PLAIN_VALUE})
    public String hello(
        @RequestParam("name")String name,
        @AuthenticationPrincipal UserDetails user) {
        String username = user==null ? null : user.getUsername();
        String greeting = "hello, " + name;
        return username==null ? greeting : greeting + " (from " + username + ")";
    }
}
```

260.2. Starting Example

We can start the web application using the Spring Boot plugin `run` goal.

Starting Example Spring Boot Web Application

```
$ mvn spring-boot:run

   _-----_
  /     \   ) )
 ( ( ) \  /  \  ) ) )
 \  \  /  \  /  \  ) ) )
  '  |  _  |  .  \  |  _  |  \  _  ,  |  /  /  /
 ======|_|=====|__/_=/_/_/_/
 :: Spring Boot ::          (2.4.2)
 ...
Tomcat started on port(s): 8080 (http) with context path ''
Started DockerHelloExampleApp in 1.972 seconds (JVM running for 2.401)
```

260.3. Client Access

Once started, we can access the `HelloController` running on localhost and the assigned 8080 port.

Accessing Local Spring Boot Web Application

```
$ curl http://localhost:8080/api/hello?name=jim  
hello, jim
```

Security is enabled, so we can also access the same endpoint with credentials and get authentication feedback.

Accessing Local Spring Boot Web Application with Credentials

```
$ curl http://localhost:8080/api/hello?name=jim -u "user:password"  
hello, jim (from user)
```

260.4. Local Unit Integration Test

The example also includes a set of unit integration tests that perform the same sort of functionality that we demonstrated with curl a moment ago.

Local Unit Integration Test

```
docker-hello-example/
`-- src/test/java/info/ejava/examples/svc
    '-- docker
        '-- hello
            |-- ClientTestConfiguration.java
            '-- HelloLocalNTest.java
    '-- resources
        '-- application-test.properties
```

Local Unit Integration Test Results

```
$ mvn clean test
10:12:54.692 main  INFO  i.e.e.svc.docker.hello.HelloLocalNTest#init:38
baseUrl=http://localhost:51319
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 4.079 s - in
info.ejava.examples.svc.docker.hello.HelloLocalNTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] -----
[INFO] BUILD SUCCESS
```

Chapter 261. Maven Heroku Deployment

When ready for application deployment, Heroku provides two primary styles of deployment with for a normal Maven application:

- git repository
- Maven plugin

The git repository requires that your deployment follow a pre-defined structure from the root—which is not flexible enough for a class demonstration tree with nested application modules. If you go that route, it may also require a separate [Procfile](#) to address startup.

The [Heroku Maven plugin](#) encapsulates everything we need to define our application startup and has no restriction on root repository structure.

261.1. Heroku Maven Plugin

The following snippets show an example use of the [Heroku Maven Plugin](#) used in this example. Documentation details are available on [GitHub](#). It has been parameterized to be able to work with most applications and is defined in the pluginDependencies section of the [ejava-build-parent](#) parent pom.xml.

Base Project Maven Properties

```
<properties>
    <java.source.version>11</java.source.version>
    <java.target.version>11</java.target.version>
    <heroku-maven-plugin.version>3.0.3</heroku-maven-plugin.version>
</properties>
```

Example Heroku Maven Plugin Configuration

```
<plugin>
  <groupId>com.heroku.sdk</groupId>
  <artifactId>heroku-maven-plugin</artifactId>
  <version>${heroku-maven-plugin.version}</version>
  <configuration>
    <jdkVersion>${java.target.version}</jdkVersion>
    <includeTarget>false</includeTarget> ①
    <includes> ②
      <include>target/${project.build.finalName}.jar</include>
    </includes>
    <processTypes> ③
      <web>java $JAVA_OPTS -jar target/${project.build.finalName}.jar
--server.port=$PORT $JAR_OPTS
      </web>
    </processTypes>
  </configuration>
</plugin>
```

- ① don't deploy entire contents of target directory
- ② identify specific artifacts to deploy; Spring Boot executable JAR
- ③ takes on role of **Procfile**; supplying the launch command

You will see mention of the **\$PORT** parameter in the [Heroku Profile documentation](#). This is a value we need to set our server port to when deployed. We can easily do that with the **--server.port** property.

To be honest, looking back on my example, I am not positive where **\$JAR_OPTS** was derived from, but know that any variables in the command line can be supplied/overridden with the **configVars** element.

261.2. Deployment appName

The deployment will require an app-name. [Heroku recommends](#) creating a profile for each of the deployment environments (e.g., development, integration, and production) and supplying the appName in those profiles. However, I am showing just a single deployment — so I set the appName separately through a property in my settings.xml.

Example appName Setting

```
<properties>
  <heroku.appName>ejava-boot</heroku.appName> ①
</properties>
```

- ① the Heroku Maven Plugin can have its **appName** set using a Maven property or element.

261.3. Example settings.xml Profile

The following shows an example of setting our `heroku appName` Maven property using `$HOME/.m2/settings.xml`. The upper `profiles` portion is used to define the profile. The lower `activeProfiles` portion can be optionally used to statically declare the profile to always be active.

Example \$HOME/.m2/settings.xml Profile

```
<?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">

  <profiles>
    <profile> ①
      <id>ejava</id>
      <properties>
        <heroku.appName>ejava-boot</heroku.appName>
      </properties>
    </profile>
  </profiles>

  <activeProfiles> ②
    <activeProfile>ejava</activeProfile>
  </activeProfiles>
</settings>
```

① defines a group of of Maven properties to be activated with a Maven profile

② profiles can be statically defined to be activated

The alternative to `activeProfiles` is to use either an activation in the pom.xml or on the command line.

Example Command Line Profile Activation

```
$ mvn (command) -Pejava
```

261.4. Using Profiles

If we went the profile route, it could look something like the following with `dev` being unique per developer and `stage` having a more consistent name across the team.

Example Use of Profiles to Set Property

```
<profiles>
  <profile>
    <id>dev</id>
    <properties> ①
      <heroku.appName>${my.dev.name}</heroku.appName>
    </properties>
  </profile>
  <profile>
    <id>stage</id>
    <properties> ②
      <heroku.appName>our-stage-name</heroku.appName>
    </properties>
  </profile>
</profiles>
```

① variable expansion based on individual `settings.xml` values when `-Pdev` profile set

② well-known-name for staging environment when `-Pstage` profile set

261.5. Maven Heroku Deploy Goal

The following shows the example output for the `heroku:deploy` Maven goal.

Example Maven heroku:deploy Goal

```
$ mvn heroku:deploy
...
[INFO] --- heroku-maven-plugin:3.0.3:deploy (default-cli) @ docker-hello-example ---
[INFO] -----> Packaging application...
[INFO]         - including: target/docker-hello-example-6.0.0-SNAPSHOT.jar
[INFO]         - including: pom.xml
[INFO] -----> Creating build...
[INFO]         - file: /var/folders/zm/cskr47zn0yjd0zwkn870y5sc0000gn/T/heroku-
deploy10792228069435401014source-blob.tgz
[INFO]         - size: 22MB
[INFO] -----> Uploading build...
[INFO]         - success
[INFO] -----> Deploying...
[INFO] remote:
[INFO] remote: -----> heroku-maven-plugin app detected
[INFO] remote: -----> Installing JDK 11... done
[INFO] remote: -----> Discovering process types
[INFO] remote:       Procfile declares types -> web
[INFO] remote:
[INFO] remote: -----> Compressing...
[INFO] remote:       Done: 81.6M
[INFO] remote: -----> Launching...
[INFO] remote:       Released v3
[INFO] remote:       https://ejava-boot.herokuapp.com/ deployed to Heroku
[INFO] remote:
[INFO] -----> Done
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 35.516 s
```

261.6. Tail Logs

We can gain some insight into the application health by tailing the logs.

```
$ heroku logs --app ejava-boot --tail
Starting process with command `--server.port=\${PORT:-8080}\'
...
Tomcat started on port(s): 54644 (http) with context path ''
Started DockerHelloExampleApp in 9.194 seconds (JVM running for 9.964)
```

261.7. Access Site

We can access the deployed application at this point using HTTPS.

Access Deployed Application on Heroku

```
$ curl -v https://ejava-boot.herokuapp.com/api/hello?name=jim
hello, jim
```

Notice that we deployed an HTTP application and must access the site using HTTPS. Heroku is providing the TLS termination without any additional work on our part.

Heroku Server Cert

```
* Server certificate:
*   subject: CN=*.herokuapp.com
*   start date: Jun 1 00:00:00 2021 GMT
*   expire date: Jun 30 23:59:59 2022 GMT
*   subjectAltName: host "ejava-boot.herokuapp.com" matched cert's "*.herokuapp.com"
*   issuer: C=US; O=Amazon; OU=Server CA 1B; CN=Amazon
*   SSL certificate verify ok.
```

261.8. Access Via Swagger

We can also access the site via swagger with a minor amount of configuration.

The screenshot shows the Swagger UI for the 'hello-controller' endpoint. At the top, there's a navigation bar with a back arrow, forward arrow, refresh button, and a URL field containing 'ejava-boot.herokuapp.com/swagger-ui/index.html?configUrl=/v3/api-docs/swagger-config#/hello-controller'. To the right of the URL is a 'Servers' dropdown set to 'https://ejava-boot.herokuapp.com - Generated server url' and an 'Authorize' button with a lock icon.

The main content area is titled 'hello-controller'. It shows a 'GET /api/hello' operation. Under 'Parameters', there is one required parameter named 'name' with type 'string' and value 'jim' (query). There are 'Execute' and 'Clear' buttons below the parameters.

Under 'Responses', there is a 'Responses' section with a 'Curl' command, a 'Request URL' (https://ejava-boot.herokuapp.com/api/hello?name=jim), and a 'Server response' section. The 'Server response' shows a status code of 200 with a response body of 'hello, jim'. There are 'Copy' and 'Download' buttons next to the response body. Below the response body, the 'Response headers' section lists several headers including 'cache-control: no-cache, no-store, max-age=0, must-revalidate', 'connection: keep-alive', 'content-length: 10', 'content-type: text/plain; charset=UTF-8', 'date: Tue, 02 Nov 2021 18:24:23 GMT', 'expires: 0', 'pragma: no-cache', and 'server: Cowboy'.

Figure 107. Access Via Swagger

To configure Swagger to ignore the injected `@AuthenticationPrincipal` parameter—we need to annotate it as hidden, using a Swagger annotation.

Eliminate Injected Parameters

```
import io.swagger.v3.oas.annotations.Parameter;
...
public String hello(
    @RequestParam("name") String name,
    @Parameter(hidden = true) //for swagger
    @AuthenticationPrincipal UserDetails user) {
```

Chapter 262. Remote IT Test

We have seen many times that there are different levels of testing that include:

- unit tests (with Mocks)
- unit integration tests (horizontal and vertical; with Spring context)
- integration tests (heavyweight process; failsafe)

No one type of test is going to be the best in all cases. In this particular case we are going to assume that all necessary unit (core functionality) and unit integration (Spring context integration) tests have been completed and we want to evaluate our application in an environment that resembles production deployment.

262.1. JUnit IT Test Case

To demonstrate the remote test, I have created a single `HelloHerokuIT` JUnit test and customized the `@Configuration` to be able to be used to express remote server aspects.

262.1.1. Test Case Definition

The failsafe integration test case looks like most unit integration test cases by naming `@Configuration` class(es), active profiles, and following a file naming convention (`IT`). The `@Configuration` is used to define beans for the IT test to act as a client of the remote server. The `heroku` profile contains properties defining identity of remote server.

Test Case Definition

```
@SpringBootTest(classes=ClientTestConfiguration.class, ①
    webEnvironment = SpringBootTest.WebEnvironment.NONE) ②
@ActiveProfiles({"test", "heroku"}) ③
public class HelloHerokuIT { ④}
```

① `@Configuration` defines beans for client testing

② no server active within JUnit IT test JVM

③ activate property-specific profiles

④ failsafe test case class name ends with IT

262.1.2. Injected Components and Setup

This specific test injects 2 users (anonymous and authenticated), the username of the authenticated user, and the `baseUrl` of the remote application. The `baseUrl` is used to define a template for the specific call being executed.

```
@Autowired  
private RestTemplate anonymousUser;  
@Autowired  
private RestTemplate authnUser;  
@Autowired  
private String authnUsername;  
private UriComponentsBuilder helloUrl;  
  
@BeforeEach  
void init(@Autowired URI baseUrl) {  
    log.info("baseUrl={}", baseUrl);  
    helloUrl = UriComponentsBuilder.fromUri(baseUrl).path("api/hello")  
        .queryParam("name", "{name}"); ①  
}
```

① `helloUrl` is a `baseUrl + /api/hello?name={name}` template

262.2. IT Properties

The integration test case pulls production properties from `src/main` and test properties from `src/test`.

262.2.1. Application Properties

The application is using a single user with its username and password statically defined in `application.properties`. These values will be necessary to authenticate with the server—even when remote.

`application.properties`

```
spring.security.user.name=user  
spring.security.user.password=password
```

Do Not Store Credentials in JAR



Do not store credentials in a resource file within the application. Resource files are generally checked into CM repositories and part of JARs published to artifact repositories. A resource file is used here to simplify the class example. A realistic solution would point the application at a protected directory or source of properties at runtime.

262.2.2. IT Test Properties

The `application-heroku.properties` file contains 3 non-default properties for the `ServerConfig`. `scheme` is hardcoded to `https`, but the `host` and `port` are defined with `${placeholder}` variables that will be filled in with Maven properties using the `maven-resources-plugin`.

- We do this for `host`, so that the `heroku.appName` can be pulled from an environment-specific properties
- We do this for `port`, to be certain that `server.http.port` is set within the `pom.xml` because the `ejava-build-parent` configures `failsafe` to pass the value of that property as `it.server.port`.

application-heroku.properties

```
it.server.scheme=https ①
it.server.host=${heroku.appName}.herokuapp.com ②
it.server.port=${server.http.port} ③ ④
```

- ① using HTTPS protocol
- ② Maven resources plugin configured to filter value during compile
- ③ Maven filtered version of property used directly within IDE
- ④ runtime failsafe configuration will provide value override

262.2.3. Maven Property Filtering

Maven copies resource files from the source tree to the target tree by default using the `maven-resources-plugin`. This plugin supports file filtering when copying files from the `src/main` and `src/test` areas. This is so common, that the `definition can be expressed outside the boundaries of the plugin`. The snippet below shows the setup of filtering a single file from `src/test/resources` and uses elements `testResource/testResources`. Filtering a file from `src/main` (not used here) would use elements `resources/resource`.

The filtering is setup in two related definitions: what we are filtering (`filtering=true`) and everything else (`filtering=false`). If we accidentally leave out the `filtering=false` definition, then only the filtered files will get copied. We could have simply filtered everything but that can accidentally destroy binary files (like images and truststores) if they happen to be placed in that path. It is safer to be explicit about what must be filtered.

Maven Property Filtering

```
<build> ①
  <testResources> <!-- used to replace ${variables} in property files -->
    <testResource> ②
      <directory>src/test/resources</directory>
      <includes> <!-- replace ${heroku.appName} -->
        <include>application-heroku.properties</include>
      </includes>
      <filtering>true</filtering>
    </testResource>
    <testResource> ③
      <directory>src/test/resources</directory>
      <excludes>
        <exclude>application-heroku.properties</exclude>
      </excludes>
      <filtering>false</filtering>
    </testResource>
  </testResources>
  ...

```

① Maven/resources-maven-plugin configured here to filter a specific file in `src/test`

② `application-heroku.properties` will be filtered when copied

③ all other files will be copied but not filtered



Maven Resource Filtering can Harm Some Files

Maven resource filtering can damage binary files and naively constructed property files (that are meant to be evaluated at runtime versus build time). It is safer to enumerate what needs to be filtered than to blindly filter all resources.

262.2.4. Property Value Sources

The source for the Maven properties can come from many places. The example sets a default within the `pom.xml`. We expect the `heroku.appName` to be environment-specific, so if you deploy the example using Maven—you will need to add `-Dheroku.appName=your-app-name` to the command line or through your local `settings.xml` file.

```
<properties> ①
  <heroku.appName>ejava-boot</heroku.appName>
  <server.http.port>443</server.http.port>
</properties>
```

① default values - can be overridden by command and `settings.xml` values

262.2.5. Maven Process Resource Phases

The following snippet shows the two resource phases being executed. Our `testResources` are copied and filtered in the second phase.

Maven Process Resource Phases

```
$ mvn clean process-test-resources
[INFO] --- maven-clean-plugin:3.1.0:clean (default-clean) @ docker-hello-example ---
[INFO] --- maven-resources-plugin:3.1.0:resources (default-resources) @ docker-hello-example ---
[INFO] --- maven-compiler-plugin:3.8.1:compile (default-compile) @ docker-hello-example ---
[INFO] --- maven-resources-plugin:3.1.0:testResources (default-testResources) @ docker-hello-example ---

$ cat target/test-classes/application-heroku.properties
it.server.scheme=https
it.server.host=ejava-boot.herokuapp.com
it.server.port=443
```

The following snippet shows the results of the property filtering using a custom value for `heroku.appName`

Maven Property Override

```
$ mvn clean process-test-resources -Dheroku.appName=other-name ①
$ cat target/test-classes/application-heroku.properties
it.server.scheme=https
it.server.host=other-name.herokuapp.com ②
it.server.port=443
```

① custom Maven property supplied on command-line

② supplied value expanded during resource filtering

262.3. Configuration

The `@Configuration` class sets up 2 RestTemplate `@Bean` factories: `anonymousUser` and `authnUser`. Everything else is there to mostly support the setup of the HTTPS connection. This same `@Configuration` is used for both the unit and failsafe integration tests. The `ServerConfig` is injected during the failsafe IT test (using `application-heroku.properties`) and instantiated locally during the unit integration test (using `@LocalPort` and default values).

ClientTestConfiguration

```
@Configuration(proxyBeanMethods = false)
@EnableConfigurationProperties //used to set it.server properties
@EnableAutoConfiguration
public class ClientTestConfiguration {
```

262.3.1. Authentication

Credentials (from `application.properties`) are injected into the `@Configuration` class using `@Value`

injection. The username for the credentials is made available as a `@Bean` to evaluate test results.

Authentication

```
@Value("${spring.security.user.name}") ①
private String username;
@Value("${spring.security.user.password}")
private String password;

@Bean
public String authnUsername() { return username; } ②
```

① default values coming from `application.properties`

② username exposed only to support evaluating authentication results

262.3.2. Server Configuration (Client Properties)

The remote server configuration is derived from properties available at runtime and scoped under the "it.server" prefix. The definitions within the `ServerConfig` instance can be used to form the baseUrl for the remote server.

ServerConfig

```
@Bean
@ConfigurationProperties(prefix = "it.server")
public ServerConfig itServerConfig() {
    return new ServerConfig();
}

//use for IT tests
@Bean ①
public URI baseUrl(ServerConfig serverConfig) {
    URI baseUrl = serverConfig.build().getBaseUrl();
    return baseUrl;
}
```

① baseUrl resolves to <https://ejava-boot.herokuapp.com:443>

262.3.3. anonymousUser

An injectable RestTemplate is exposed with no credentials as "anonymousUser". As with most of our tests, the `BufferingClientHttpRequestFactory` has been added to support multiple reads required by the `RestTemplateLoggingFilter` (which provides debug logging). The `ClientHttpRequestFactory` was made injectable to support HTTP/HTTPS connections.

anonymousUser

```
@Bean  
public RestTemplate anonymousUser(RestTemplateBuilder builder,  
                                  ClientHttpRequestFactory requestFactory) { ①  
    return builder.requestFactory(  
        //used to read the streams twice ③  
        ()->new BufferingClientHttpRequestFactory(requestFactory))  
    .interceptors(new RestTemplateLoggingFilter()) ②  
    .build();  
}
```

① `requestFactory` will determine whether HTTP or HTTPS connection created

② `RestTemplateLoggingFilter` provides HTTP debug statements

③ `BufferingClientHttpRequestFactory` caches responses, allowing it to be read multiple times

262.3.4. authnUser

An injectable `RestTemplate` is exposed with valid credentials as "authnUser". This is identical to `anonymousUser` except credentials are provided through a `BasicAuthenticationInterceptor`.

authnUser

```
@Bean  
public RestTemplate authnUser(RestTemplateBuilder builder,  
                           ClientHttpRequestFactory requestFactory) {  
    return builder.requestFactory(  
        //used to read the streams twice  
        ()->new BufferingClientHttpRequestFactory(requestFactory))  
    .interceptors(  
        new BasicAuthenticationInterceptor(username, password), ①  
        new RestTemplateLoggingFilter())  
    .build();  
}
```

① valid credentials added

262.3.5. ClientHttpRequestFactory

The builder requires a `requestFactory` and we have already shown that it will be wrapped in a `BufferingClientHttpRequestFactory` to support debug logging. However, the core communications is implemented by the `org.apache.http.client.HttpClient` class.

ClientHttpRequestFactory

```
import org.apache.http.client.HttpClient;
import org.apache.http.impl.client.HttpClientBuilder;
import javax.net.ssl.SSLContext;
...
@Bean
public ClientHttpRequestFactory httpsRequestFactory(
    ServerConfig serverConfig, ①
    SSLContext sslContext) { ②
    HttpClient httpsClient = HttpClientBuilder.create()
        .setSSLContext(serverConfig.isHttps() ? sslContext : null)
        .build();
    return new HttpComponentsClientHttpRequestFactory(httpsClient);
}
```

① `ServerConfig` provided to determine whether HTTP or HTTPS required

② `SSLContext` provided for when HTTPS is required

262.3.6. SSL Context

The `SSLContext` is provided by the `org.apache.http.ssl.SSLContextBuilder` class. In this particular instance, we expect the deployment environment to use commercial, trusted certs. This will eliminate the need to load a custom truststore.

```
import org.apache.http.ssl.SSLContextBuilder;
import javax.net.ssl.SSLContext;
...
@Bean
public SSLContext sslContext(ServerConfig serverConfig) {
    try {
        URL trustStoreUrl = null;
        //using trusted certs, no need for customized truststore
        //...
        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);
    }
}
```

262.4. JUnit IT Test

The following shows two sanity tests for our deployed application. They both use a base URL of <https://ejava-boot.herokuapp.com/api/hello?name={name}> and supply the request-specific `name` property through the `UriComponentsBuilder.build(args)` method.

262.5. Simple Communications Test

When successful, the simple communications test will return a 200/OK with the text "hello, jim"

Simple Communications Test

```
@Test
void can_contact_server() {
    //given
    String name="jim";
    URI url = helloUrl.build(name);
    RequestEntity<Void> request = RequestEntity.get(url).build();
    //when
    ResponseEntity<String> response = anonymousUser.exchange(request, String.class);
    //then
    then(response.getStatusCode()).isEqualTo(HttpStatus.OK);
    then(response.getBody()).isEqualTo("hello, " + name); ①
}
```

① "hello, jim"

262.6. Authentication Test

When successful, the authentication test will return a 200/OK with the text "hello, jim (from user)". The name for "user" will be the username injected from the `application.properties` file.

Authentication Test

```
@Test
void can_authenticate_with_server() {
    //given
    String name="jim";
    URI url = helloUrl.build(name);
    RequestEntity<Void> request = RequestEntity.get(url).build();
    //when
    ResponseEntity<String> response = authnUser.exchange(request, String.class);
    //then
    then(response.getStatusCode()).isEqualTo(HttpStatus.OK);
    then(response.getBody()).isEqualTo("hello, " +name+ " (from " +authnUsername+ ")"
);①
}
```

① "hello, jim (from user)"

262.7. Automation

The IT tests have been disabled to avoid attempts to automatically deploy the application in every build location. Automation can be enabled at two levels: test and deployment.

262.7.1. Enable IT Test

We can enable the IT tests alone by adding `-DskipITs=value`, where `value` is anything but `true`, `false`, or blank.

- `skipITs` (blank) and `skipITs=true` will cause failsafe to not run. This is a standard failsafe behavior.
- `skipITs=false` will cause the application to be re-deployed to Heroku. This is part of our custom `pom.xml` definition that will be shown in a moment.

Execute IT Test Only

```
$ mvn verify -DitOnly -DskipITs=not_true ① ② ③
...
GET https://ejava-boot.herokuapp.com:443/api/hello?name=jim, returned OK/200
hello, jim
...
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
[INFO] -----
[INFO] BUILD SUCCESS
```

① `verify` goal completes the IT test phases

② `itOnly` - defined by `ejava-build-parent` to disable surefire tests

③ `skipITs` - controls whether IT tests are performed



skipITs can save Time and Build-time Dependencies

Setting `skipITs=true` can save time and build-time dependencies when all that is desired is a resulting artifact produced by `mvn install`.

262.7.2. Enable Heroku Deployment

The pom also has conditionally added the `heroku:deploy` goal to the `pre-integration` phase if `skipITs=false` is explicitly set. This is helpful if changes have been made. However, know that a full upload and IT test execution is a significant amount of time to spend. Therefore, it is not the thing one would use in a rapid test, code, compile, test repeat scenario.

Enable Heroku Deployment

```
<profiles>
  <profile> <!-- deploys a Spring Boot Executable JAR -->
    <id>heroku-it-deploy</id>
    <activation>
      <property> ①
        <name>skipITs</name>
        <value>false</value>
      </property>
    </activation>
    <properties> ②
      <spring-boot.repackage.skip>false</spring-boot.repackage.skip>
    </properties>
    <build>
      <plugins>
        <plugin> ③
          <groupId>com.heroku.sdk</groupId>
          <artifactId>heroku-maven-plugin</artifactId>
          <executions>
            <execution>
              <id>deploy</id>
              <phase>pre-integration-test</phase>
              <goals>
                <goal>deploy</goal>
              </goals>
            </execution>
          </executions>
        </plugin>
      </plugins>
    </build>
  </profile>
</profiles>
```

① only one of `skipITs` has the value `false`

② make sure that JAR is a Spring Boot executable JAR

③ add deploy step in `pre-integration` phase

IT Test Results with Heroku Deploy

```
$ mvn verify -DitOnly -DskipITs=false
...
[INFO] --- spring-boot-maven-plugin:2.4.2:repackage (package) @ docker-hello-example
---
[INFO] Replacing main artifact with repackaged archive
[INFO] <<< heroku-maven-plugin:3.0.3:deploy (deploy) < package @ docker-hello-example
<<<
[INFO] --- heroku-maven-plugin:3.0.3:deploy (deploy) @ docker-hello-example ---
[INFO] jakarta.el-3.0.3.jar already exists in destination.

...
[INFO] -----> Done
[INFO] --- maven-failsafe-plugin:3.0.0-M5:integration-test (integration-test) @
docker-hello-example ---
...
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
[INFO] --- maven-failsafe-plugin:3.0.0-M5:verify (verify) @ docker-hello-example ---
[INFO] -----
[INFO] BUILD SUCCESS
```

Chapter 263. Summary

In this module we learned:

- to deploy an application under development to Heroku cloud provider to make it accessible to Internet users
 - using naked Spring Boot form
- to deploy incremental and iterative changes to the application
- how to interact with your developed application on the Internet

Docker Images

copyright © 2021 jim stafford (jim.stafford@jhu.edu)

Chapter 264. Introduction

We have seen where we already have many of the tools we need to be able to develop, test, and deploy a functional application. However, there will become a point where things will get complicated.

- What if everything is not a Spring Boot application and requires a unique environment?
- What if you end up with dozens of applications and many versions?
 - Will everyone on your team be able to understand how to instantiate them?

Lets take a user-level peek at the Docker container in order to create a more standardized look to all our applications.

264.1. Goals

You will learn:

- the purpose of an application container
- to identify some open standards in the Docker ecosystem
- to build a Docker images using different techniques
- to build a layered Docker image

264.2. Objectives

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

1. build a basic Docker image with an executable JAR using a Dockerfile and docker commands
2. build a basic Docker image with the Spring Boot Maven Plugin and buildpack
3. build a layered Docker image with the Spring Boot Maven Plugin and buildpack
4. build a layered Docker image using a Dockerfile and docker commands
5. run a docker image hosting a Spring Boot application

Chapter 265. Containers

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. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.

— docker.com, "What is a Container" A standardized unit of software

265.1. Container Deployments

The following diagrams represent three common application deployment strategies: native, virtual machine, and container.

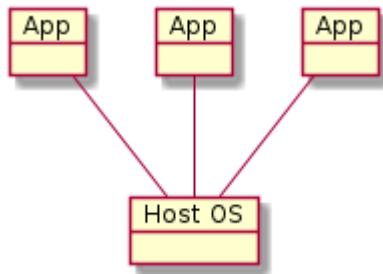


Figure 108. Native Deployment

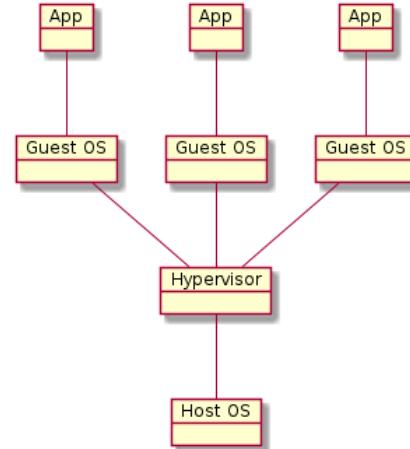


Figure 109. VM Deployment

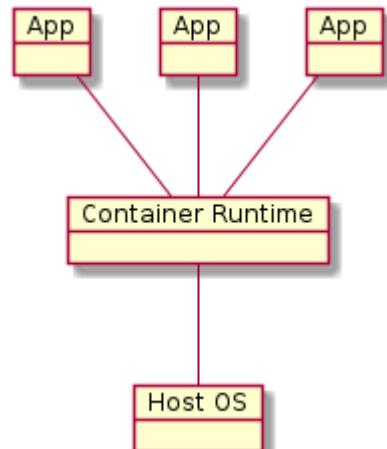


Figure 110. Container Deployment

- **native** - has the performance advantage of running on bare metal but the disadvantage of having full deployment details exposed and the vulnerability of directly sharing the same host operating system with other processes.
- **virtual machine** - (e.g., VMWare, VirtualBox) has the advantage of isolation from other processes and potential encapsulation of installation details but the disadvantage of a separate and distinct guest operating systems running on the same host with limited sharing of resources.
- **container** - has the advantage of isolation from other processes, encapsulation of installation details, and runs in a lightweight container runtime that efficiently shares the resources of the host OS with each container.

Chapter 266. Docker Ecosystem

Docker is an ecosystem of tooling that covers a lot of topics. Two of which are the container image and runtime. The specifications of both of these have been initiated by Docker—the company—and transitioned to the Open Container Initiative (OCI)—a standards body—that maintains the definition of the image and runtime specs and certifications.

This has allowed independent toolsets (for building Docker images) and runtimes (for running Docker images under different runtime and security conditions). For example, the following is a sample of the alternative builders and runtimes available.

266.1. Container Builders

Docker—the company—offers a Docker image builder. However, the builder requires a daemon with a root-level installation. Some of the following simply implement a builder tool:

- [buildah](#)
- [ocibuilder](#)
- [orca-build](#)
- [jessfraz/img](#)

266.2. Container Runtimes

Docker—the company—offers a container runtime. However, this container runtime has a complex lifecycle that includes daemons and extra processes. Some of the following simply run an image.

- [podman](#)
- [kata containers](#)
- [Windows Hyper-V Containers](#)
- [cri-o](#)

Chapter 267. Docker Images

A Docker image is a tar file of layered, intermediate levels of the application. A layer within a Docker image contains a tar file of the assigned artifacts for that layer. If two or more Docker files share the same base layer—there is no need to repeat the base layer in that repository. If we change the upper levels of a Docker file, there is no need to rebuild the lower levels. These aspects will be demonstrated within this lecture and optimized in the tooling available to use within Spring Boot.

Chapter 268. Basic Docker Image

We can build a basic Docker image from a normal executable JAR created from the Spring Boot Maven Plugin. To prove that—we will return to the [hello-docker-example](#) used in the previous Heroku deployment lecture.



Example Requires Docker Installed

Implementing the first example will require docker—the product—to be installed. Please see the [development environment Docker setup](#) for references.

The following shows us starting with a typical example web application that listens to port 8080 when built and launched. The build happens to automatically invoke the [spring-boot:repackage](#) goal. However, if that is not the case, just run `mvn spring-boot:repackage` to build the Spring Boot executable JAR.

Building and Running Basic Docker Image

```
$ mvn clean package ①
...
target/
|-- [ 25M] docker-hello-example-6.0.0-SNAPSHOT.jar
|-- [5.4K] docker-hello-example-6.0.0-SNAPSHOT.jar.original

$ java -jar target/docker-hello-example-6.0.0-SNAPSHOT.jar ②
...
Tomcat started on port(s): 8080 (http) with context path ''
Started DockerHelloExampleApp in 3.058 seconds (JVM running for 3.691)
```

① building the executable Spring Boot JAR

② running the application

268.1. Basic Dockerfile

We can build a basic Docker image manually by adding a Dockerfile and issuing a Docker command to build it.

The basic Dockerfile below extends a base OpenJDK 14 image from the global Docker repository, adds the executable JAR, and registers the default commands to use when running the image. It happens to have the name [Dockerfile.execjar](#), which will be referenced by a later command.

Example Basic Dockerfile (named Dockerfile.execjar)

```
FROM adoptopenjdk:14-jre-hotspot ①
COPY target/*.jar application.jar ②
ENTRYPOINT ["java", "-jar", "application.jar"] ③
```

① building off a base openjdk 14 image

- ② copying executable JAR into the image
- ③ establishing default command to run the executable JAR

268.2. Basic Docker Image Build Output

The Docker build command processes the Dockerfile and produces an image. We supply the Dockerfile, the directory `(.)` of the source files referenced by the Dockerfile, and an image name and tag.

Example docker build Command Output

```
$ docker build . -f Dockerfile.execjar -t docker-hello-example:execjar ① ② ③ ④
Sending build context to Docker daemon 26.06MB
Step 1/3 : FROM adoptopenjdk:14-jre-hotspot
--> 8e632ae8b66e
Step 2/3 : COPY target/*.jar application.jar
--> 4ce5f2c555f5
Step 3/3 : ENTRYPOINT ["java", "-jar", "application.jar"]
--> Running in 0be7aa3e08b2
Removing intermediate container 0be7aa3e08b2
--> eda93db54671
Successfully built eda93db54671
Successfully tagged docker-hello-example:execjar
```

- ① `build` - command to build Docker image
- ② `.` - current directory is default source
- ③ `-f` - path to Dockerfile, if not `Dockerfile` in current directory
- ④ `name:tag` - name and tag of image to create

Dockerfile is default name for Dockerfile



Default Docker file name is `Dockerfile`. This example will use multiple Dockerfiles, so the explicit `-f` naming has been used.

268.3. Local Docker Registry

Once the build is complete, the image is available in our local repository with the name and tag we assigned.

Example Local Repository

```
$ docker images | egrep 'docker-hello-example|REPO'
REPOSITORY          TAG      IMAGE ID      CREATED       SIZE
docker-hello-example  execjar  eda93db54671  12 minutes ago  293MB
```

- `REPOSITORY` - names the primary name of the Docker image
- `TAG` - primarily used to identify versions and variants of repository name. `latest` is the default

tag

- IMAGE ID - is a hex string value that identifies the image. The repository:tag label just happens to point to that version right now, but will advance in a future change/build.
- SIZE - is total size if exported. Since Docker images are layered, multiple images sharing the same base image will supply much less overhead than reported here while staged in a repository

268.4. Running Docker Image

We can run the image with the `docker run` command. The following example shows running the `docker-hello-image` with tag `execjar`, exposing port 8080 within the image as port 9090 on localhost (`-p 9090:8080`), running in interactive mode (`-it`; optional here, but important when using as interactive shell), and removing the runtime image when complete (`--rm`).

Example Docker Run Command

```
$ docker run --rm -it -p 9090:8080 docker-hello-example:execjar ① ② ③ ④
 .-----⑤
 /\\ / ____'_ __ _ `|_( )_ ___` __ _` \ \ \ \
( ( )\__ | ' | ' | | ' \ \ \ ' | \ \ \ \
\ \ \ __) | ( )| | | | | || ( | | ) ) ) )
 ' |_____| .__| | | | | \_, | / / /
=====|_|=====|_|/_=/|/_/_/
 :: Spring Boot ::           (2.4.2)
...
Tomcat started on port(s): 8080 (http) with context path ''
Started DockerHelloExampleApp in 4.049 seconds (JVM running for 4.784)
```

① `run` - run a command in a new Docker image

② `--rm` - remove the image instance when complete

③ `-it` allocate a pseudo-TTY (`-t`) for an interactive (`-i`) shell

④ `-p` - map external port `9090` to `8080` of the internal process

⑤ Spring Boot App launched with no arguments

268.5. Docker Run Command with Arguments

Arguments can also be passed into the image. The example below passes in a standard Spring Boot property to turn off printing of the startup banner.

Example Docker Run Command with Arguments

```
$ docker run --rm -it -p 9090:8080 docker-hello-example:execjar --spring.main.banner
-mode=off
...
Tomcat started on port(s): 8080 (http) with context path ''
Started DockerHelloExampleApp in 4.049 seconds (JVM running for 4.784)
```

① `spring.main.banner-mode` property passed to Spring Boot App and disabled banner printing

268.6. Running Docker Image

We can verify the process is running using the Docker `ps` command.

Example docker ps Command

```
$ docker ps
CONTAINER ID   IMAGE               COMMAND                  CREATED
STATUS         PORTS              NAMES
8078f6369a59  docker-hello-example:execjar "java -jar applicati..." 4 minutes ago
Up 4 minutes   0.0.0.0:9090->8080/tcp   practical_agnesi
```

- CONTAINER ID - hex string we can use to refer to this running (or later terminated) instance
- IMAGE - REPO:TAG executed
- COMMAND - command executed upon entry
- CREATED - when started
- STATUS - run status. Use `docker ps -a` to locate all images and not just running images
- PORTS - lists ports exposed within image and what they are mapped to externally on the host
- NAMES - textual name alias for instance. Can be used interchangeably with containerId. Can be explicitly set with `--name foo` option prior to the image parameter, but must be unique

268.7. Using the Docker Image

We can call our Spring Boot process within the image using the mapped 9090 port.

```
$ curl http://localhost:9090/api/hello?name=jim
hello, jim
```

268.8. Docker Image is Layered

The Docker image is a TAR file that is made up of layers

Example Docker Image Tarfile Contents

```
$ docker save docker-hello-example:execjar > image.tar
Mac:image$ tar tf image.tar
27dcc15ccaaac941791ba5826356a254e70c85d4c9c8954e9c4eb2873506a4c8/
27dcc15ccaaac941791ba5826356a254e70c85d4c9c8954e9c4eb2873506a4c8/VERSION
27dcc15ccaaac941791ba5826356a254e70c85d4c9c8954e9c4eb2873506a4c8/json
27dcc15ccaaac941791ba5826356a254e70c85d4c9c8954e9c4eb2873506a4c8/layer.tar
304740117a5a0c15c8ea43b7291479207b357b9fc08cc47a5e4a357f5e9a1768/
304740117a5a0c15c8ea43b7291479207b357b9fc08cc47a5e4a357f5e9a1768/VERSION
304740117a5a0c15c8ea43b7291479207b357b9fc08cc47a5e4a357f5e9a1768/json
304740117a5a0c15c8ea43b7291479207b357b9fc08cc47a5e4a357f5e9a1768/layer.tar
...
a3651512f2a9241ae11ad8498df67b4f943ea4943f4fae8f88bcb0b81168803d/
a3651512f2a9241ae11ad8498df67b4f943ea4943f4fae8f88bcb0b81168803d/VERSION
a3651512f2a9241ae11ad8498df67b4f943ea4943f4fae8f88bcb0b81168803d/json
a3651512f2a9241ae11ad8498df67b4f943ea4943f4fae8f88bcb0b81168803d/layer.tar
...
manifest.json
repositories
```

This specific example has seven (7) layers.

Example Layer Count

```
$ tar tf image.tar | grep layer.tar | wc -l
7
```

268.9. Application Layer

If we untar the Docker image and poke around, we can locate the layer that contains our executable JAR file. All 25M of it in one place.

Example Application Layer

```
$ tar tf ./a3651512f2a9241ae11ad8498df67b4f943ea4943f4fae8f88bcb0b81168803d/layer.tar
application.jar ①

ls -lh ./a3651512f2a9241ae11ad8498df67b4f943ea4943f4fae8f88bcb0b81168803d/layer.tar
25M ./a3651512f2a9241ae11ad8498df67b4f943ea4943f4fae8f88bcb0b81168803d/layer.tar
```

① one of the layers contains our application layer and is made up of a single Spring Boot executable JAR

There are a few things to note about what we uncovered in this section

1. the Docker image is not a closed, binary representation. It is an openly accessible layer of files as defined by the [OCI Image Format Specification](#).
2. our application is currently implemented as a **single** 25MB layer with a **single** Spring Boot

executable JAR. Our code was likely only a few KBytes of that 25MB.

Hold onto both of those points when covering the next topic.

268.10. Spring Boot Plugin

Starting with Spring Boot 2.3 and its enhanced support for cloud technologies, the Spring Boot Maven Plugin now provides support for building a Docker image using [buildpack](#)—not Docker and no Dockerfile.

```
$ mvn spring-boot:help  
...  
spring-boot:build-image  
  Package an application into a OCI image using a buildpack.
```

Buildpack is an approach to building Docker images based on strict layering concepts that Docker has always prescribed. The main difference with buildpack is that the layers are more autonomous—backed by a segment of industry—allowing for higher level application layers to be quickly rebased on top of patched operating system layers without fully rebuilding the image.

[Joe Kutner from Heroku stated](#) at a Spring One Platform conference that they were able to patch 10M applications overnight when a serious bug was corrected in a base layer. This was due to being able to rebase the application specific layers with a new base image using buildpack technology and without having to rebuild the images. ^[54]

268.11. Building Docker Image using Buildpack

If we look at the portions of the generated output, we will see

- 15 candidate buildpacks being downloaded
- one of the 5 used buildpacks is specific to spring-boot
- various layers are generated and reused to build the image
- our application still ends up in a single layer
- the image is generated, by default using the Maven artifactId as the image name and version number as the tag

Example Maven Building using Buildpack

```
$ mvn clean spring-boot:build-image
...
[INFO] --- spring-boot-maven-plugin:2.3.2.RELEASE:build-image (default-cli) @ docker-hello-example ---
[INFO] Building image 'docker.io/library/docker-hello-example:6.0.0-SNAPSHOT'
[INFO]
[INFO] > Pulling builder image 'gcr.io/paketo-buildpacks/builder:base-platform-api-0.3' 6%
...
[INFO] > Pulling builder image 'gcr.io/paketo-buildpacks/builder:base-platform-api-0.3' 100%
[INFO] > Pulled builder image 'gcr.io/paketo-buildpacks/builder@sha256:6d625fe00a2b5c4841eccb6863ab3d8b6f83c3138875f48ba69502abc593a62e'
[INFO] > Pulling run image 'gcr.io/paketo-buildpacks/run:base-cnb' 100%
[INFO] > Pulled run image 'gcr.io/paketo-buildpacks/run@sha256:087a6a98ec8846e2b8d75ae1d563b0a2e0306dd04055c63e04dc6172f6ff6b9d'
'
[INFO] > Executing lifecycle version v0.8.1
[INFO] > Using build cache volume 'pack-cache-2432a78c0232.build'
[INFO]
[INFO] > Running creator
[INFO]   [creator]    ==> DETECTING
[INFO]   [creator]    5 of 16 buildpacks participating
...
[INFO]   [creator]    paketo-buildpacks/spring-boot      2.4.1
...
[INFO]   [creator]    ==> EXPORTING
[INFO]   [creator]    Reusing layer 'launcher'
[INFO]   [creator]    Adding layer 'paketo-buildpacks/bellsoft-liberica:class-counter'
[INFO]   [creator]    Reusing layer 'paketo-buildpacks/bellsoft-liberica:java-security-properties'
...
[INFO]   [creator]    Adding 1/1 app layer(s)
[INFO]   [creator]    Adding layer 'config'
[INFO]   [creator]    *** Images (10a764b20812):
[INFO]   [creator]          docker.io/library/docker-hello-example:6.0.0-SNAPSHOT
[INFO]
[INFO] Successfully built image 'docker.io/library/docker-hello-example:6.0.0-SNAPSHOT'
```

268.12. Buildpack Image in Local Docker Repository

The newly built image is now installed into the local Docker registry. It is using the Maven GAV artifactId for the repository and version for the tag.

Docker Repository with both Images

```
$ docker images | egrep 'docker-hello-example|IMAGE'  
REPOSITORY          TAG      IMAGE ID      CREATED        SIZE  
docker-hello-example  execjar  eda93db54671  40 minutes ago  315MB ①  
docker-hello-example  6.0.0-SNAPSHOT  10a764b20812  41 years ago   279MB ①
```

① NOTE: sizes were from a later build using newer versions of Spring Boot



One odd thing is the timestamp used (41 years ago) for the created date with the build pack image. Since it is referring to the year 1970 (new java.util.Date(0) UTC), we can likely assume there was a 0 value in a timestamp field somewhere.

268.13. Buildpack Image Execution

Notice that when we run the newly built image that was built with buildpack, we get a little different behavior at the beginning where some base level memory tuning is taking place.

Example Buildpack Image Execution

```
$ docker run --rm -it -p 9090:8080 docker-hello-example:6.0.0-SNAPSHOT  
Container memory limit unset. Configuring JVM for 1G container.  
Calculated JVM Memory Configuration: -XX:MaxDirectMemorySize=10M  
-XX:MaxMetaspaceSize=87032K -XX:ReservedCodeCacheSize=240M -Xss1M -Xmx449543K (Head  
Room: 0%, Loaded Class Count: 12952, Thread Count: 250, Total Memory: 1.0G)  
Adding 127 container CA certificates to JVM truststore  
Spring Cloud Bindings Boot Auto-Configuration Enabled  
...  
Tomcat started on port(s): 8080 (http) with context path ''  
Started DockerHelloExampleApp in 3.589 seconds (JVM running for 4.3)
```

The following shows we are able to call the new running image.

Example Buildpack Image Call

```
$ curl http://localhost:9090/api/hello?name=jim  
hello, jim
```

268.14. Inspecting Buildpack Image

If we save off the newly build image and briefly inspect, we will see that it contains the same TAR-based layering scheme but will 21 versus 7 layers in this specific example.

Buildpack Layer Count

```
$ docker save docker-hello-example:6.0.0-SNAPSHOT > image.tar
$ tar tf image.tar | grep layer.tar | wc -l
21
```

If we untar the mage and poke around, we can eventually locate our application and notice that it happens to be in exploded form versus executable JAR form. We can see our code and dependency libraries separately.

Buildpack Application Layer

```
$ tar tf 6e2b5eb3b4b11627cce2ca7c8aeb7de68a7a54b56b15ea4d43e4a14d2b1f0b9a/layer.tar
...
/workspace/BOOT-
INF/classes/info/ejava/examples/svc/docker/hello/DockerHelloExampleApp.class
/workspace/BOOT-
INF/classes/info/ejava/examples/svc/docker/hello/controllers/ExceptionAdvice.class
/workspace/BOOT-
INF/classes/info/ejava/examples/svc/docker/hello/controllers/HelloController.class
...
/workspace/BOOT-INF/lib/classgraph-4.8.69.jar
/workspace/BOOT-INF/lib/commons-lang3-3.10.jar
/workspace/BOOT-INF/lib/ejava-dto-util-6.0.0-SNAPSHOT.jar
/workspace/BOOT-INF/lib/ejava-util-6.0.0-SNAPSHOT.jar
/workspace/BOOT-INF/lib/ejava-web-util-6.0.0-SNAPSHOT.jar
```

As a reminder, when we built the Docker image with a Docker file and vanilla docker commands—we ended up with an application layer with a single, Spring Boot executable JAR (with a few KBytes of our code and 24.9 MB of dependency artifacts).

Review: Earlier Generic Docker Image Application Layer

```
$ tar tf ./a3651512f2a9241ae11ad8498df67b4f943ea4943f4fae8f88bcb0b81168803d/layer.tar
application.jar

ls -lh ./a3651512f2a9241ae11ad8498df67b4f943ea4943f4fae8f88bcb0b81168803d/layer.tar
25M ./a3651512f2a9241ae11ad8498df67b4f943ea4943f4fae8f88bcb0b81168803d/layer.tar
```

[54] "Pack to the Future: Cloud-Native Buildpacks on k8s", Spring One Platform, Oct 2019

Chapter 269. Layers

Dockerfile layers are an important concept when it comes to efficiency of storage and distribution. Any images built on common base images or intermediate commands that produce the same result do not have to be replicated within a repository. For example, 100 images all extending from the same OpenJDK 14 image do not need to have the OpenJDK 14 portions repeated.

To make it easier to view and analyze the layers of the Dockerfile — we can use a simple inspection tool called [dive](#). This shows us how the image is constructed, where we may have wasted space, and potentially how to optimize. Since these images are brand new and based off production base images — we will not see much wasted space at this time. However, it will help us better understand the Docker image and how cloud features added to Spring Boot can help us.

Dive Not Required



There is no need to install the dive tool to learn about layers and how Spring Boot provides support for layers. All necessary information to understand the topic is contained in the following material.

Running dive on Docker Image

```
$ dive [imageId or name:tag]
```

With the image displayed, I find it helpful to:

- hit [CNTL]+L if "Show Layer Changes is not yet selected"
- hit [TAB] to switch to "Current Layer Contents" pane on the right
- hit [CNTL]+U,R,M, and B to turn off all display except "Added"
- hit [TAB] to switch back to "Layers" pane on the left

In the "Layers" pane we can scroll up and down the layers to see which files were added because of which ordered command in the Dockerfile. If all the layers look the same, make sure you are only displaying the "Added" artifacts.

Dive within Docker

Or — of course — you could run dive within Docker to inspect a Docker image. This requires that you map the image's Docker socket to the host machine's Docker socket with the `-v` syntax. This is likely OS-specific.



```
docker run --rm -it -v /var/run/docker.sock:/var/run/docker.sock wagoodman/dive [imageId or name:tag]
```

269.1. Analyzing Basic Docker Image

In this first example, we are looking at the layers of the basic Dockerfile. Notice:

- a majority of the size was the result of extending the OpenJDK image. That space represents content that a Dockerfile **repository does not have to replicate**.
- the last layer contains the 26MB executable JAR. Because that technically contains our custom application. This is content a Dockerfile **repository has to replicate**.

Analyzing Basic Docker Image

```
$ dive docker-hello-example:execjar
```

Layers			Current Layer Contents			
Cmp	Size	Command	Permission	UID:GID	Size	Filetree
	63 MB	FROM 304740117a5a0c1	-rw-r--r--	0:0	26 MB	application.jar
	988 kB	[-z "\$(apt-get indextargets)"]				
	745 B	set -xe && echo '#!/bin/sh' > /usr/sbin/policy-rc.d &				
	7 B	mkdir -p /run/systemd && echo 'docker' > /run/systemd/co				
	36 MB	apt-get update && apt-get install -y --no-install-re				
	167 MB	set -eux; ARCH="\$(dpkg --print-architecture)"; c				
	26 MB	#(nop) COPY file:576755947381c122473841c08cc3454bd7f1bcc				

Layer Details

Tags: (unavailable)
Id: a3651512f2a9241ae11ad8498df67b4f943ea4943f4fae8f88bcb0b81168
803d
Digest: sha256:3c53295d85fea258cb2ceef882a4a608c003b0bf6638aef285b9f
4c0a9b4742b
Command:
#(nop) COPY file:576755947381c122473841c08cc3454bd7f1bcc6ee8999db78f
e8a3d2a81d6de in application.jar

Image Details

Total Image size: 293 MB
Potential wasted space: 3.0 MB
Image efficiency score: 99 %

Count	Total Space	Path
2	1.3 MB	/var/cache/debconf/templates.dat
2	562 kB	/var/cache/apt/pkgcache.bin
2	425 kB	/var/cache/apt/srcpkgcache.bin
2	405 kB	/var/log/dpkg.log
2	212 kB	/var/lib/dpkg/status

Actions: ^C Quit | Tab Switch view | ^F Filter | ^L Show layer changes | ^A Show aggregated changes

269.2. Analyzing Basic Buildpack Image

If we look at the Docker image built with buildpack, through the Maven plugin, we will see the same 26MB exploded as separate files towards the end of the image. From a layering perspective—the exploded structure has not saved us anything.

Analyzing Buildpack Image

```
$ dive docker-hello-example:6.0.0-SNAPSHOT
```

The screenshot shows the Docker Compose interface with two main sections: 'Layers' and 'Current Layer Contents'.

Layers:

Cmp	Size	Command
	63 MB	FROM 304740117a5a0c1
	988 kB	
	745 B	
	7 B	
	225 B	
	20 MB	
	398 kB	
	2.3 MB	
	6.7 MB	
	214 B	
	140 MB	
	3.0 MB	
	7.2 MB	
	2.2 MB	
	7.2 MB	
	7.0 MB	
	10 B	
	53 kB	
	3 B	
26 MB		
	15 kB	

Layer Details:

- Tags: (unavailable)
- Id: 6e2b5eb3b4b11627cce2ca7c8aeb7de68a7a54b56b15ea4d43e4a14d2b1f0b9a
- Digest: sha256:2c6cc70a3550435f3ea6b90c6e8895410dc45f13b9ef875db16df1f5c7ab0d38
- Command:

Current Layer Contents:

```

● Current Layer Contents
└── workspace
    ├── BOOT-INF
    │   └── classes
    │       └── application.properties
    ├── info
    │   └── ejava
    │       └── examples
    │           └── svc
    │               └── docker
    │                   └── hello
    │                       ├── DockerHelloExampleApp.class
    │                       └── controllers
    │                           ├── ExceptionAdvice.class
    │                           └── HelloController.class
    └── classpath.idx
    └── lib
        ├── classgraph-4.8.69.jar
        ├── commons-lang3-3.10.jar
        ├── ejava-dto-util-6.0.0-SNAPSHOT.jar
        ├── ejava-util-6.0.0-SNAPSHOT.jar
        ├── ejava-web-util-6.0.0-SNAPSHOT.jar
        ├── jackson-annotations-2.11.1.jar
        ├── jackson-core-2.11.1.jar
        ├── jackson-databind-2.11.1.jar
        ├── jackson-dataformat-xml-2.11.1.jar
        ├── jackson-dataformat-yaml-2.11.1.jar
        ├── jackson-datatype-jdk8-2.11.1.jar
        ├── jackson-datatype-jsr310-2.11.1.jar
        ├── jackson-module-jaxb-annotations-2.11.1.jar
        ├── jackson-module-parameter-names-2.11.1.jar
        ├── jakarta.activation-api-1.2.2.jar
        ├── jakarta.annotation-api-1.3.5.jar
        └── jakarta.el-3.0.3.jar

```

Bottom navigation bar: ^C Quit | Tab Switch view | ^F Filter | Space Collapse dir | ^Space Collapse all dir | ^A Added | ^R Removed | ^M Modified | ^U Unmodified | ^B A

However, now that we have it exploded — we will have the option to break it into further layers.

Chapter 270. Adding Fine-grain Layering

Having all 26MB of our Spring Boot application in a single layer can be wasteful—especially if we push new images to a repository many times during development. We end up with 26MB.version1, 26MB.version2, etc. when each push is more than likely a few modifications of class files within the application and a complete change in library dependencies not as common.

270.1. Configure Layer-ready Executable JAR

The Spring Boot plugin and buildpack provide support for creating finer-grain layers from the executable JAR by enabling the `layers` plugin configuration property.

```
<plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
  <configuration>
    <layers>
      <enabled>true</enabled>
    </layers>
  </configuration>
</plugin>
```

270.2. Building and Inspecting Layer-ready Executable JAR

If we rebuild the executable JAR with the layered option, an extra wrapper is added to the executable JAR file that can be activated with the `-Djarmode=layeredtools` option to the `java -jar` command. This option takes one of two arguments: list or extract.

Inspecting Layer-ready Executable JAR

```
$ mvn clean package spring-boot:repackage -Dlayered=true -DskipTests ①
$ java -Djarmode=layeredtools -jar target/docker-hello-example-6.0.0-SNAPSHOT.jar
Usage:
  java -Djarmode=layeredtools -jar docker-hello-example-6.0.0-SNAPSHOT.jar
```

Available commands:

- list List layers from the jar that can be extracted
- extract Extracts layers from the jar for image creation
- help Help about any command

① `-Dlayered=true` activates layering within the Maven pom.xml

270.3. Default Executable JAR Layers

Spring Boot automatically configures four (4) layers by default: (released) dependencies, spring-boot-loader, snapshot-dependencies, and application. These layers are ordered from most stable (dependencies) to least stable (application). We have the ability to change the layers—but I won't go into that here.

Default Executable JAR Layers

```
$ java -Djarmode=layer-tools -jar target/docker-hello-example-6.0.0-SNAPSHOT.jar list  
dependencies  
spring-boot-loader  
snapshot-dependencies  
application
```

Chapter 271. Layered Buildpack Image

With the `layers` configuration property enabled, the next build will result in a layered image posted to the local Docker repository.

```
$ mvn package spring-boot:build-image -Dlayered=true -DskipTests  
...  
Successfully built image 'docker.io/library/docker-hello-example:6.0.0-SNAPSHOT'
```

271.1. Dependency Layer

The dependency layer contains all the released dependencies. This happens to make up most of the 26MB we had for the executable JAR. This 26MB **does not need to be replicated** in the image repository if consistent with follow-on publications of our image.

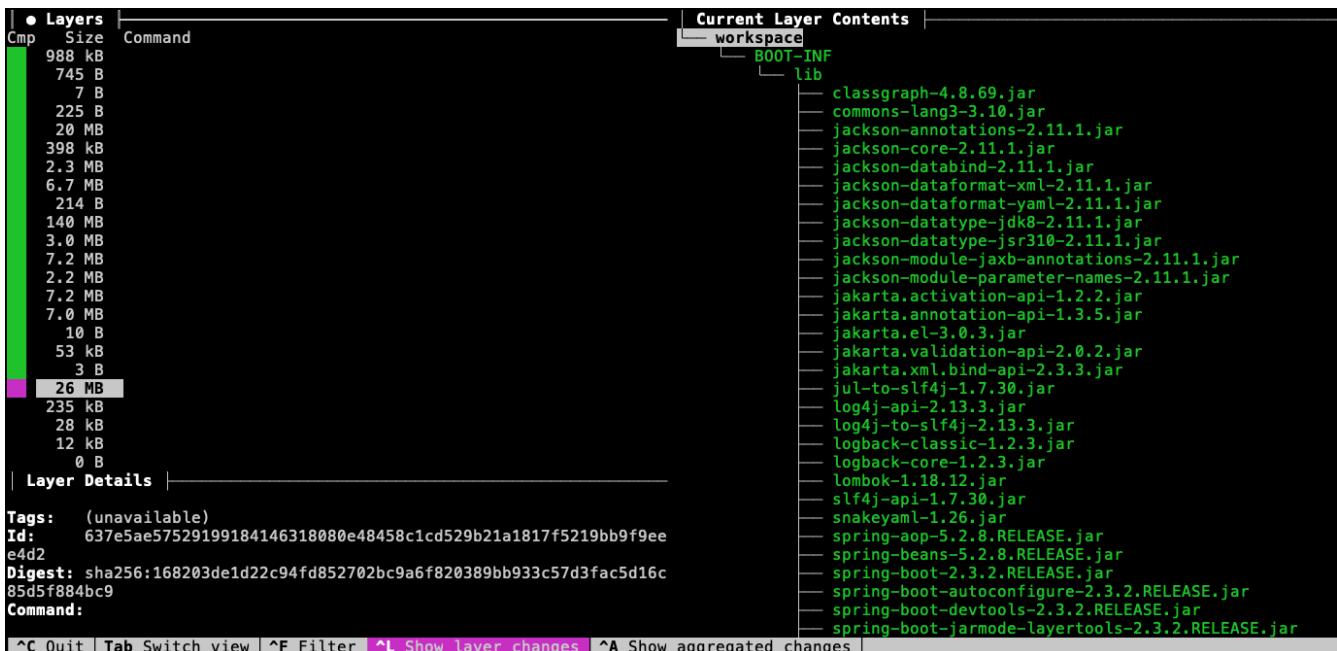


Figure 111. Dependency Layer

271.2. Snapshot Layer

The snapshot layer contains dependency artifacts that have not been released. This is an indication that the artifact is slightly more stable than our application code but not as stable as the released dependencies.

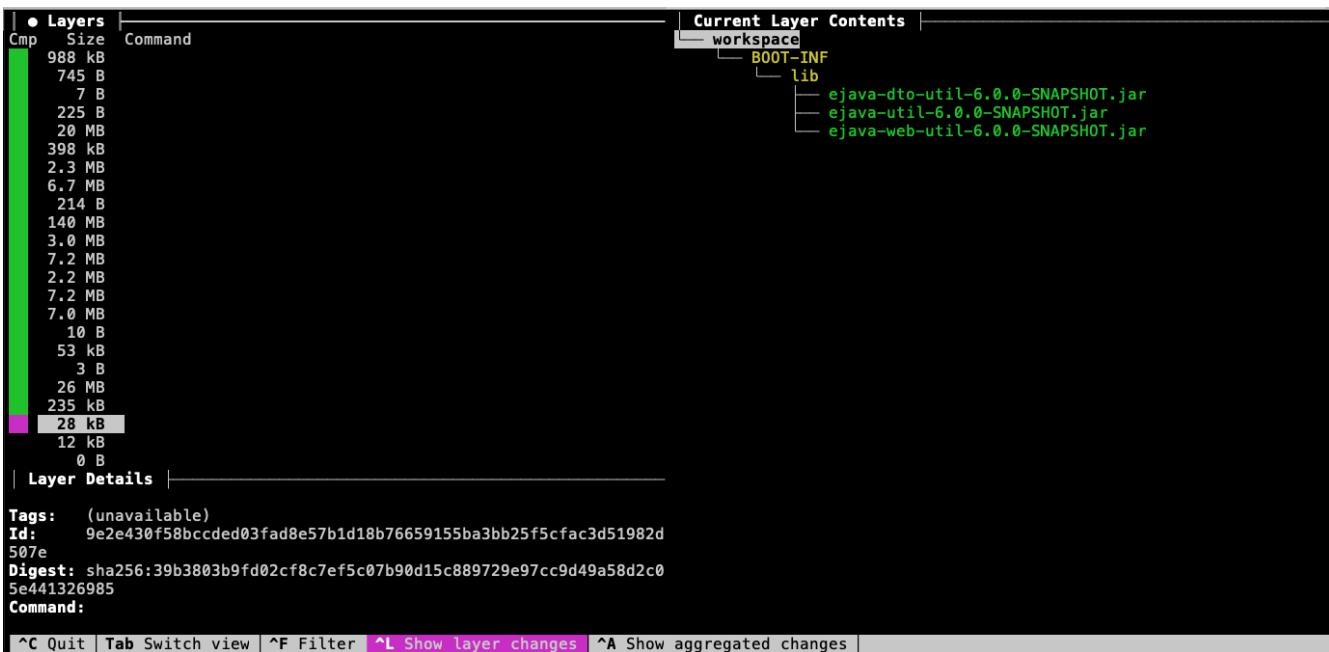


Figure 112. Snapshot Dependency Layer

271.3. Application Layer

The application layer contains the code for the local module—which should be the most volatile. Notice that in this example, the application code is 12KB out of the total 26MB for the executable JAR. If we change our application code and redeploy the image somewhere—only this small portion of the code needs to change.

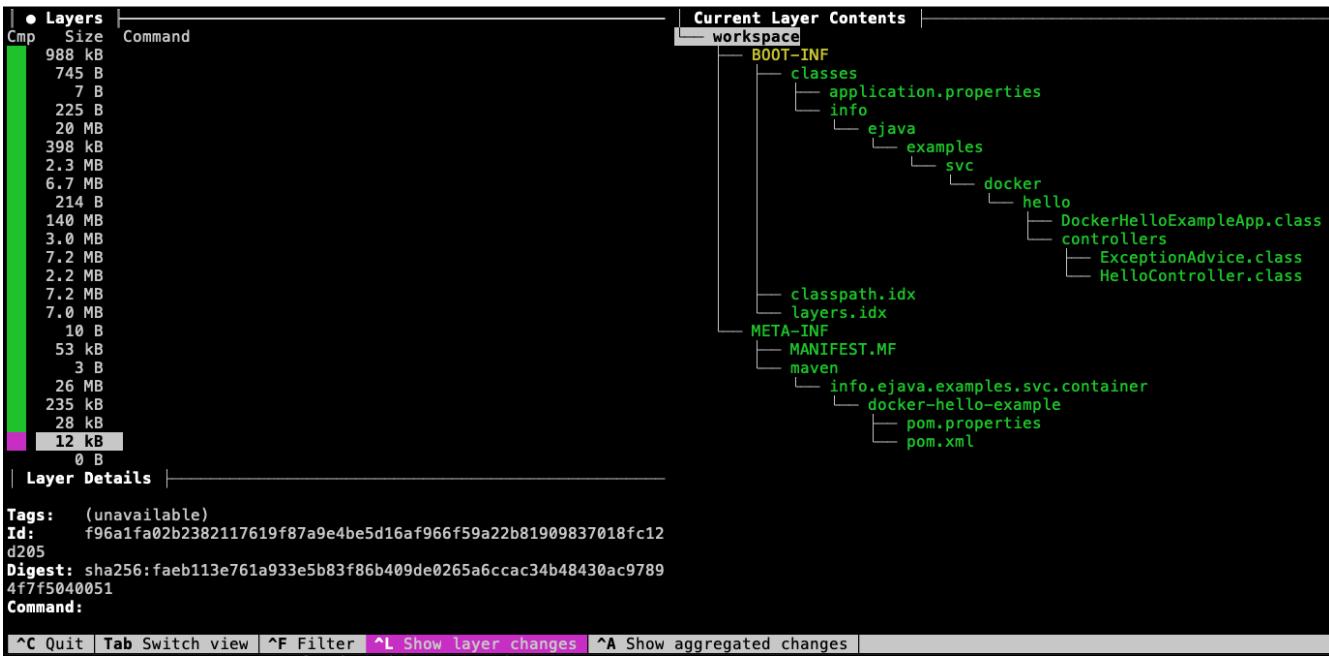


Figure 113. Application Layer

271.4. Review: Single Layer Application

If you remember ... before we added multiple layers, all the library stable JARs and semi-stable SNAPSHOT dependencies were in the same layers as our potentially changing application code. We now have them in separate layers.

Review: Single Layer Application

```
└── workspace
    └── BOOT-INF
        ├── classes
        │   ├── application.properties
        │   ├── info
        │   │   └── ejava
        │   │       └── examples
        │   │           └── svc
        │   │               └── docker
        │   │                   └── hello
        │   │                       ├── DockerHelloExampleApp.class
        │   │                       └── controllers
        │   │                           ├── ExceptionAdvice.class
        │   │                           └── HelloController.class
        ├── classpath.idx
        └── lib
            ├── classgraph-4.8.69.jar
            ├── commons-lang3-3.11.jar
            ├── ejava-dto-util-6.0.0-SNAPSHOT.jar
            └── ejava-util-6.0.0-SNAPSHOT.jar
```

Chapter 272. Layered Docker Image

Since buildpack may not be for everyone, Spring Boot provides a means for standard Docker users to create layered images with a standard Dockerfile and standard `docker` commands. The following example is based on the [Example Dockerfile](#) on the Spring Boot features page.

272.1. Example Layered Dockerfile

The Dockerfile is written in two parts: builder and image construction. The first, builder half of the file copies in the executable JAR and extracts the layer directories into a temporary portion of the image.

The second, construction half builds the final image by extending off what could be an independent parent image and the products of the builder phase. Notice how the four (4) layers are copied in separately - forming distinct boundaries.

Example Layered Dockerfile

```
FROM adoptopenjdk:14-jre-hotspot as builder ①
WORKDIR application
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} application.jar
RUN java -Djarmode=layer tools -jar application.jar extract

FROM adoptopenjdk:14-jre-hotspot 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/ ./
ENTRYPOINT ["java", "org.springframework.boot.loader.JarLauncher"]
```

① commands used to setup building the image

② commands used to build the image

Example Layered Dockerfile Build

```
$ docker build . -f Dockerfile.layered -t docker-hello-example:layered
Sending build context to Docker daemon 26.1MB
```

272.2. Builder Phase

The following shows the output of building our example using the `docker build` command and the Dockerfile above. Notice that it copies in the executableJAR and extracts the layers into the temporary image.

Example Docker Image Builder Phase

```
Step 1/12 : FROM adoptopenjdk:14-jre-hotspot as builder
--> 8e632ae8b66e
Step 2/12 : WORKDIR application
--> Using cache
--> 7b628ccd3023
Step 3/12 : ARG JAR_FILE=target/*.jar
--> Using cache
--> 8c6c7e49ed62
Step 4/12 : COPY ${JAR_FILE} application.jar
--> 0e8dcdef3cfc
Step 5/12 : RUN java -Djarmode=layer-tools -jar application.jar extract
--> Running in 44eaade21110
Removing intermediate container 44eaade21110
--> 113512ce6506
```

272.3. Construction Phase

The next set of output is from the construction phase of the `docker build` command. Notice how it is building separate, distinct layers by using separate COPY commands for each layer directory.

Example Docker Image Construction Phase

```
Step 6/12 : FROM adoptopenjdk:14-jre-hotspot
--> 8e632ae8b66e
Step 7/12 : WORKDIR application
--> Using cache
--> 7b628ccd3023
Step 8/12 : COPY --from=builder application/dependencies/ ./
--> Using cache
--> 4d245588baf9
Step 9/12 : COPY --from=builder application/spring-boot-loader/ ./
--> Using cache
--> 3231757e1e93
Step 10/12 : COPY --from=builder application/snapshot-dependencies/ ./
--> a27c34c98f33
Step 11/12 : COPY --from=builder application/application/ ./
--> 03f759997e26
Step 12/12 : ENTRYPOINT ["java", "org.springframework.boot.loader.JarLauncher"]
--> Running in 48f90ae3e625
Removing intermediate container 48f90ae3e625
--> f1ba667557ba
Successfully built f1ba667557ba
Successfully tagged docker-hello-example:layered
```

272.4. Dependency Layer

The dependency layer — like with the buildpack version — contains 26MB of the released JARs. This makes up the bulk of what was in our executable JAR.

This screenshot shows the Docker image layers interface. The left pane displays the 'Layers' table with the following data:

Cmp	Size	Command
63 MB	FROM 304740117a5a0c1	
988 kB	[-z "\$(apt-get indextargets)"]	
745 B	set -xe && echo '#!/bin/sh' > /usr/sbin/policy-rc.d &	
7 B	mkdir -p /run/systemd && echo 'docker' > /run/systemd/co	
36 MB	apt-get update && apt-get install -y --no-install-re	
167 MB	set -eux; ARCH=\$(dpkg --print-architecture); c	
0 B	#(nop) WORKDIR /application	
26 MB	#(nop) COPY dir:ca3f1f80ba03c0af675d3905cf8af20e7f5c977	
235 kB	#(nop) COPY dir:637c983b7f385801c12135959479cf2d23d3dc52	
28 kB	#(nop) COPY dir:f097ea2816ffbe677a84610f5828252b889f5f99	
12 kB	#(nop) COPY dir:90786121f1e4f876210f835651a4173659e1fc3a	

The right pane shows the 'Current Layer Contents' for the 'application' layer, specifically the 'BOOT-INF/lib' directory, which contains numerous JAR files including classgraph, commons-lang, jackson annotations, jackson core, jackson databind, jackson dataformat xml, jackson dataformat yaml, jackson datatype jdk8, jackson datatype jsr310, jackson module jaxb annotations, jackson module parameter names, jakarta activation api, jakarta annotation api, jakarta el, jakarta validation api, jakarta xml bind api, jul-to-slf4j, log4j api, log4j to slf4j, logback classic, logback core, lombok, slf4j api, snakeyaml, spring aop, spring beans, spring boot, spring boot autoconfigure, spring boot devtools, spring boot jarmode layertools, and spring boot jarmode layertools.

272.5. Snapshot Layer

The snapshot layer contains dependencies that have not yet been released. These are believed to be more stable than our application code but less stable than the released dependencies.

This screenshot shows the Docker image layers interface. The left pane displays the 'Layers' table with the following data:

Cmp	Size	Command
63 MB	FROM 304740117a5a0c1	
988 kB	[-z "\$(apt-get indextargets)"]	
745 B	set -xe && echo '#!/bin/sh' > /usr/sbin/policy-rc.d &	
7 B	mkdir -p /run/systemd && echo 'docker' > /run/systemd/co	
36 MB	apt-get update && apt-get install -y --no-install-re	
167 MB	set -eux; ARCH=\$(dpkg --print-architecture); c	
0 B	#(nop) WORKDIR /application	
26 MB	#(nop) COPY dir:ca3f1f80ba03c0af675d3905cf8af20e7f5c977	
235 kB	#(nop) COPY dir:637c983b7f385801c12135959479cf2d23d3dc52	
28 kB	#(nop) COPY dir:f097ea2816ffbe677a84610f5828252b889f5f99	
12 kB	#(nop) COPY dir:90786121f1e4f876210f835651a4173659e1fc3a	

The right pane shows the 'Current Layer Contents' for the 'application' layer, specifically the 'BOOT-INF/lib' directory, which contains three JAR files: ejava-dto-util-6.0.0-SNAPSHOT.jar, ejava-util-6.0.0-SNAPSHOT.jar, and ejava-web-util-6.0.0-SNAPSHOT.jar.

272.6. Application Layer

The application layer contains our custom application code. This layer is thought to be the most volatile and is in the top-most layer.

Layers

Cmp	Size	Command
63 MB	FROM 304740117a5a0c1	
988 kB	[-z "\$(apt-get indextargets)"]	
7 kB	set -xe && echo '#!/bin/sh' > /usr/sbin/policy-rc.d &	
36 kB	mkdir -p /run/systemd && echo 'docker' > /run/systemd/co	
167 kB	apt-get update && apt-get install -y --no-install-re	c
0 kB	set -eux; ARCH="\$(dpkg --print-architecture)"; c	
26 kB	#(nop) COPY dir:ca3f1f80ba03c0afd675d3905cf8af20e7f5c977	
235 kB	#(nop) COPY dir:637c983b7f385801c12135959479cf2d23d3dc52	
28 kB	#(nop) COPY dir:f097ea2816ffbe677a84610f5828252b889f5f99	
12 kB	#(nop) COPY dir:90786121f1e4f876210f835651a4173659e1fc3a	

Layer Details

```

Tags: (unavailable)
Id: 51b81c3833813a79c10198180c8c5c4b42693f66a79d428ad72a62583155
62f7
Digest: sha256:43232323dc1d908b23318b24298e363d118f338b4bbd1ba9f1840
618aca0ec6b
Command:
#(nop) COPY dir:90786121f1e4f876210f835651a4173659e1fc3a798e60f7df42
8d9ed4c0356a in ./

```

Image Details

```

Total Image size: 293 MB
Potential wasted space: 3.0 MB
Image efficiency score: 99 %

```

Count	Total Space	Path
2	1.3 MB	/var/cache/debconf/templates.dat

Current Layer Contents

```

application
├── BOOT-INF
│   └── classes
│       └── application.properties
├── info
│   └── ejava
│       └── examples
│           └── svc
│               └── docker
│                   └── hello
│                       └── DockerHelloExampleApp.class
│   └── controllers
│       └── ExceptionAdvice.class
│   └── HelloController.class
└── classpath.idx
└── layers.idx
└── META-INF
└── MANIFEST.MF
└── maven
    └── info.ejava.examples.svc.container
        └── docker-hello-example
            └── pom.properties
            └── pom.xml

```

Commands:

- ^C Quit
- Tab Switch view
- ^F Filter
- ^L Show layer changes
- ^A Show aggregated changes

Chapter 273. Summary

In this module we learned:

- Docker is a ecosystem of concepts, tools, and standards
- Docker — the company — provides an implementation of those concepts, tools, and standards
- Docker images can be created using different tools and technologies
 - the `docker build` command uses a Dockerfile
 - buildpack uses knowledgeable inspection of the codebase
- Docker images have ordered layers — from common operating system to custom application
- buildpack layers are rigorous enough that they can be rebased upon freshly patched images — making hundreds to millions of image patches feasible within a short amount of time
- intelligent separation of code into layers and proper ordering can lead to storage and complexity savings
- Spring Boot provides a means to separate the executable JAR into layers that match certain criteria

Heroku Docker Deployments

copyright © 2021 jim stafford (jim.stafford@jhu.edu)

Chapter 274. Introduction

With a basic introduction to Docker under our belt, I would like to return to the Heroku deployment topic to identify the downside of deploying full applications—whether they be

- naked Spring Boot executable JAR
 - Spring Boot executable JAR wrapped in a Docker image
- and show the benefit of using a layered application.

This is a follow-on lecture

It is assumed that you have already covered the [Heroku deployment](#) and [Docker](#) lectures, have a Heroku account, already deployed a Spring Boot application, and interacted with that application via the Internet. If not, you will need to go back to that lecture and review the basics of getting started with the Heroku account.



If you do not have Docker—the product—installed, you should still be able to follow along to pick up the concepts.

274.1. Goals

You will learn:

- to deploy a Docker-based image to a cloud provider to make it accessible to Internet users

274.2. Objectives

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

1. make a Heroku-deployable Docker image that accepts environment variable(s)
2. deploy a Docker image to Heroku using docker repository commands
3. deploy a Docker image to Heroku using CLI commands

Chapter 275. Heroku Docker Notes

The following are Heroku references for Spring Boot and Docker deployments

- [Developing with Docker](#)
- [Working with Spring Boot](#)

Of important note — the Maven Spring Boot Plugin built Docker image (using buildpack) — uses an internal memory calculator that initially mandates 1GB of memory. This exceeds the free 512MB Heroku limit. Deploying this version of the application will immediately fail until we locate a way to change that value. However, we can successfully deploy the standard Dockerfile version — which lacks an explicit, up-front memory requirement.

We will also need to do some property expression gymnastics that will be straight forward to implement using the standard Dockerfile approach.

Chapter 276. Heroku Login

With the Heroku CLI installed—we need to login. This will redirect us to the browser where we can complete the login.

Heroku Command Line Login

```
$ heroku login  
heroku: Press any key to open up the browser to login or q to exit:  
Opening browser to https://cli-auth.herokuapp.com/auth/cli/browser/f944d777-93c7-40af-  
b772-0a1c5629c609  
Logging in... done  
Logged in as ...
```

276.1. Heroku Container Login

Heroku requires an additional login step to work with containers. With the initial login complete—no additional credentials will be asked for but this step seems required.

Additional Heroku Container Login

```
$ heroku container  
7.54.0  
$ heroku container:login  
Login Succeeded
```

276.2. Create Heroku App

At this point you are ready to again perform a one-time (per deployment app) process that will reserve an app-name for you on herokuapp.com. We know that this name is used to reference our application and form a URL to access it on the Internet.

Example Create Heroku App

```
$ heroku create [app-name] ①  
  
Creating ⚡ [app-name]... done  
https://app-name.herokuapp.com/ | https://git.heroku.com/app-name.git
```

① if app-name not supplied, a random app-name will be generated



Heroku also uses Git repositories for deployment

Heroku creates a Git repository for the app-name that can also be leveraged as a deployment interface. I will not be covering that option.

You can create more than one heroku app and the app can be renamed with the following `apps:rename` command.

Example Rename Heroku App

```
$ heroku apps:rename --app oldname newname
```

Visit the [Heroku apps](#) page to locate technical details related to your apps.

Chapter 277. Adjust Dockerfile

Heroku requires the application accept a `$PORT` environment variable to identify the listen port at startup. We know from our lessons in configuration, we can accomplish that by supplying a Spring Boot property on the command line.

Example Startup Specification of Listen Port

```
java -jar (app).jar --server.port=$PORT
```

Since we are launched using a Dockerfile and the parameter will require a shell evaluation, we can accomplish this by using the Dockerfile `CMD` command below—which will feed the `ENTRYPOINT` command its resulting values when expressed this way.^[55] I have also added a default value of 8080 when the `$PORT` variable has not been supplied (i.e., in local environment).

Example Spring Boot Dockerfile ENTRYPOINT

```
ENV PORT=8080 ①  
ENTRYPOINT ["java", "org.springframework.boot.loader.JarLauncher"] ②  
CMD ["--server.port=${PORT}"] ③
```

① default value used if `PORT` is not supplied

② `ENTRYPOINT` always executes no matter if a parameter is supplied

③ `CMD` expresses a default when no parameter(s) are supplied

277.1. Test Dockerfile server.port

We can test the configuration locally using the following commands.

277.1.1. Testing \$PORT CMD With Environment Variable

In this iteration, we are simulating the Heroku container supplying a `PORT` environment variable with a value. This value will be used by the Spring Boot application running within the Docker image. The `PORT` value is also mapped to an external `9090` value so we can call the server from the outside.

Testing With Environment Variable

```
$ mvn package spring-boot:repackage -Dlayered=true ①  
  
$ docker build . -f Dockerfile.layered -t docker-hello-example:layered  
$ docker run --rm -p 9090:5000 -e PORT=5000 docker-hello-example:layered ②  
...  
Tomcat started on port(s): 5000 (http) with context path '' ③  
Started DockerHelloExampleApp in 3.623 seconds (JVM running for 4.392)
```

① example sets layered false by default and toggles with `layered` property

② -e option defines **PORT** environment variable to value **5000**

③ --server.port=\${PORT} sees that value and server listens on port **5000**

We can use the following to test.

Testing \$PORT CMDs Mapped to 9090

```
$ curl http://localhost:9090/api/hello?name=jim  
hello, jim
```

277.1.2. Testing without Environment Variable

In this iteration, we are simulating local development independent of the Heroku container by not supplying a **PORT** environment variable and using the default from the Docker **CMD** setting. Like before, this value will be used by the Spring Boot application running within the Docker image and that value will again be mapped to external port **9090** value so we can call the server from the outside.

Testing \$PORT CMD Without Environment Variable

```
$ docker run --rm -p 9090:8080 docker-hello-example:layered ①  
Tomcat started on port(s): 8080 (http) with context path '' ②  
Started DockerHelloExampleApp in 4.414 seconds (JVM running for 5.177)
```

① no **PORT** environment variable is expressed

② server uses assigned ENV default of 8080

We can again use the following to test.

Testing \$PORT CMDs Mapped to 9090

```
$ curl http://localhost:9090/api/hello?name=jim  
hello, jim
```

[55] "Docker RUN vs CMD vs ENTRYPPOINT", Yuri Pitsishin, April 2016

Chapter 278. Deploy Docker Image

I will demonstrate two primary ways deploy a Docker image to Heroku:

1. using `docker push` command to deploy a tagged image to the Heroku Docker repository
2. using the `heroku container:push` command to build and upload an image

Both require a follow-on `heroku container:release` command to complete the deployment.

278.1. Deploying Tagged Image

One way to deploy a Docker image to Heroku is to create a Docker tag associated with the target Heroku repository and then push that image to the Docker repository. The tag has the following format

```
registry.heroku.com/[app-name]/web ①
```

① `registry.heroku.com` is the actual address of the Heroku Docker repository

My examples will use the app-name `ejava-docker`.

278.1.1. Tagging the Image

There are at least two ways to tag the image:

- WAY 1: tag the Docker image during the build

Tag Docker Image During Build

```
docker build . -f Dockerfile.layered -t registry.heroku.com/ejava-docker/web
Sending build context to Docker daemon 26.1MB
Step 1/13 : FROM adoptopenjdk:14-jre-hotspot as builder
...
Successfully tagged registry.heroku.com/ejava-docker/web:latest
```

- WAY 2: tag an existing Docker image

Tag Existing Docker Image

```
$ docker build . -f Dockerfile.layered -t docker-hello-example:layered
$ docker tag docker-hello-example:layered registry.heroku.com/ejava-docker/web
```

In either case, we will end up with a tag in the repository that will look like the following.

Example Docker Image Repository with Tagged Image

```
$ docker images | grep heroku
REPOSITORY           TAG      IMAGE ID      CREATED       SIZE
registry.heroku.com/ejava-docker/web  latest   72fe4327f05f  15 minutes ago  293MB
```

278.1.2. Deploying the Image

The last step in deploying the tagged image is to invoke `docker push` using the full name of the tag.

Push Tagged Docker Image

```
$ docker push registry.heroku.com/ejava-docker/web
The push refers to repository [registry.heroku.com/ejava-docker/web]
3e974fa6054f: Pushed
...
7ef368776582: Layer already exists
latest: digest:
sha256:37c99a899b26f2cfb192cd42f930120b11bb56408eb3e4590dfe78b957f2acf1 size: 2621
```

278.2. Push using Heroku CLI

The other alternative is to use `heroku container:push` to build and push the Docker image without going through the local repository.

```
$ heroku container:push web --app ejava-docker
```

container:push requires Dockerfile to be named Dockerfile — no file references

The `heroku container:push` command requires the Dockerfile be called `Dockerfile` and in the current directory. The command does not allow us to reference a unique filename (e.g., `Dockerfile.layered`). I used a soft link to get around that (i.e., `ln -s Dockerfile.layered Dockerfile`). The `container:push` documentation does infer that files normally referenced locally by the Dockerfile can be in a referenced location—possibly allowing the Dockerfile to be placed in a unique location versus having a unique name.



Chapter 279. Complete Deployment

A successfully pushed image will not be made immediately available. We must follow through with a `release` command.

279.1. Release Pushed Image to Users

The following command finishes the deployment — making the updated image accessible to users.

Release Pushed Image to Users

```
$ heroku container:release web --app ejava-docker
Releasing images web to ejava-docker... done
```

279.2. Tail Logs

We can gain some insight into the application health by tailing the logs.

```
$ heroku logs --app ejava-docker --tail
Starting process with command `--server.port=\$\{PORT:-8080\}`
...
Tomcat started on port(s): 54644 (http) with context path ''
Started DockerHelloExampleApp in 9.194 seconds (JVM running for 9.964)
```

279.3. Access Site

We can access the deployed application at this point but will be required to use HTTPS. Notice, however, HTTPS is fully setup with a trusted certificate.

Access Deployed Application on Heroku

```
$ curl -v https://ejava-docker.herokuapp.com/api/hello?name=jim
*   Trying 52.73.83.132...
* TCP_NODELAY set
* Connected to ejava-docker.herokuapp.com (52.73.83.132) port 443 (#0)
* SSL connection using TLSv1.2 / ECDHE-RSA-AES128-GCM-SHA256
* ALPN, server did not agree to a protocol
* Server certificate:
*   subject: C=US; ST=California; L=San Francisco; O=Heroku, Inc.; CN=*.herokuapp.com
*   start date: Jun 15 00:00:00 2020 GMT
*   expire date: Jul 7 12:00:00 2021 GMT
*   subjectAltName: host "ejava-docker.herokuapp.com" matched cert's "*.herokuapp.com"
*   issuer: C=US; O=DigiCert Inc; OU=www.digicert.com; CN=DigiCert SHA2 High Assurance
Server CA
*   SSL certificate verify ok.
> GET /api/hello?name=jim HTTP/1.1
> Host: ejava-docker.herokuapp.com
> User-Agent: curl/7.64.1
> Accept: */*
>
< HTTP/1.1 200
< Server: Cowboy
Hello, jim
```

Chapter 280. Summary

In this module we learned:

- to deploy an application under development to Heroku cloud provider to make it accessible to Internet users
 - using Docker form
- to deploy incremental and iterative changes to the application

Docker Compose

copyright © 2021 jim stafford (jim.stafford@jhu.edu)

Chapter 281. 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.

281.1. 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

281.2. 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 282. 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.

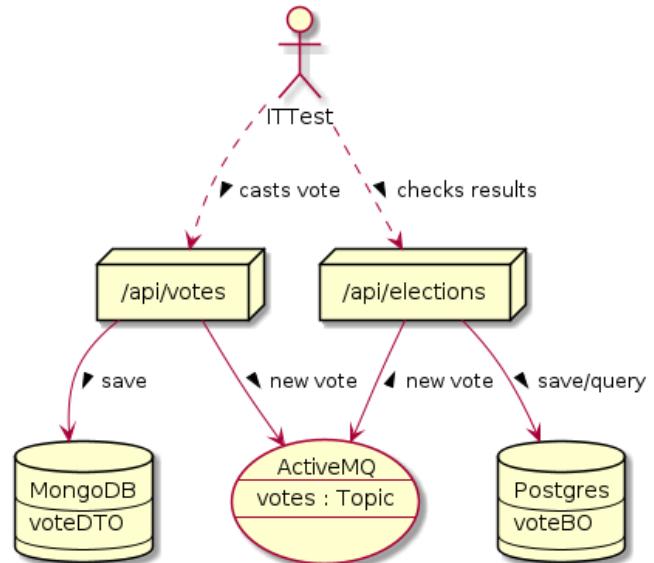


Figure 114. Need to Integrate with Specific Real Services

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.

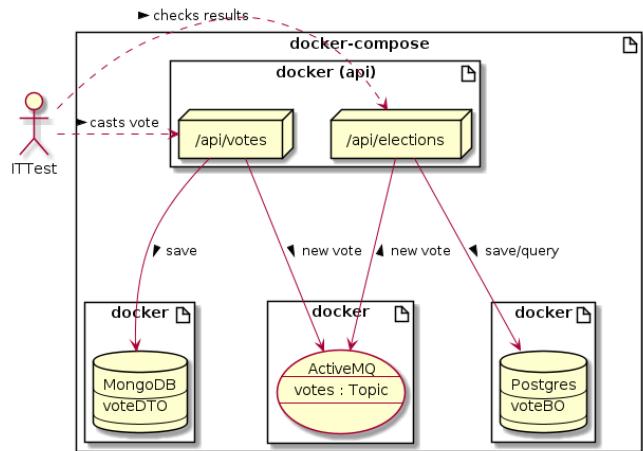


Figure 115. Virtualize Services with Docker

282.1. 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 MONGODB_URI=... \ ③
-e DATABASE_URL=... \
docker-hello-example:6.0.0-SNAPSHOT
```

① using the mongo container from Dockerhub

② using the postgres container from Dockerhub

③ using our example Spring Boot Web application; it does not yet use the databases

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 machine 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 283. Docker Compose

[DockerCompose](#) is a tool for defining and running multi-container Docker applications. With Docker Compose, we can:

- define our network of applications in a single YAML file
- start/stop applications according to defined dependencies
- run commands inside of running images
- treat the running applications as normal, running Docker images

283.1. 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 configuration files
- is the opportunity to apply environment variables and extensions
- are commands to build and control lifecycle actions of the network

Let's start with the Docker Compose configuration file.

Chapter 284. 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 a file `version` and individual `services`.

docker-compose.yml Shell

```
version: '3.8'  
services:  
  mongo:  
    ...  
  postgres:  
    ...  
  api:  
    ...
```

- **version** - informs the docker-compose binary what features could be present within the file. I have shown a recent version of `3.8` but our use of the file will be very basic and could likely be set to `3` or as low as `2`.
- **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.

284.1. 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"  
  #    - "27017:27017"  
  #    - "37001:27017"  
  #    - "127.0.0.1:37001:27017"
```

① not assigning port# here

- **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 (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 add 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

284.2. 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.

284.3. api Service Definition

The **api** service defines our API server with the Votes and Elections Services. This service will become a client of the other three services.

api Service Definition

```
api:  
  build:  
    context: .  
    dockerfile: Dockerfile.layered  
    image: docker-hello-example:layered  
  ports:  
    - "${API_PORT:-8080}:8080"  
  depends_on:  
    - mongo  
    - postgres  
  environment:  
    - spring.profiles.active=integration  
    - MONGODB_URI=mongodb://admin:secret@mongo:27017/votes_db?authSource=admin  
    - DATABASE_URL=postgres://postgres:secret@postgres:5432/postgres
```

- **build** - identifies a source Dockerfile that can build the image for this service
 - **context** - defines the path to the Dockerfile
 - **dockerfile** - defines the specific name of the Dockerfile (optional in this case)
- **image** - identifies the name and tag used for the built image
- **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 **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. 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.
 - using **spring.profiles.active** to instruct API to use **integration** profile
 - API is not yet using the databases, but these URLs are consistent with what will be encountered when deployed to Heroku.
 - 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

284.4. 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.

```
$ docker-compose build
postgres uses an image, skipping
mongo uses an image, skipping
Building api
[+] Building 0.2s (13/13) FINISHED
=> => naming to docker.io/library/docker-hello-example:layered
..
```

After the first start, a re-build is only performed using the `build` command or when the `--build` option.

284.5. Default Port Assignments

If we start the services ...

```
$ export API_PORT=1234 && docker-compose up -d ①
Creating network "docker-hello-example_default" with the default driver
Creating docker-hello-example_mongo_1    ... done
Creating docker-hello-example_postgres_1 ... done
Creating docker-hello-example_api_1      ... done
```

① `up` starts service and `-d` runs the container in the background as a daemon

You will notice that no ports were assigned to the unassigned `mongo` and `postgres` services. However, the given shown port# in the output is available to the other hosts within that Docker network. If we don't need `mongo` or `postgres` accessible to the host's network — we are good. The `api` service was assigned a variable (value `1234`) port# — which is accessible to the host's network.

```
$ docker-compose ps
      Name        State     Ports
-----
docker-hello-example_api_1    Up      0.0.0.0:1234->8080/tcp,:::1234->8080/tcp
docker-hello-example_mongo_1   Up      27017/tcp
docker-hello-example_postgres_1 Up      5432/tcp
```

Using Variable-Assigned API Port#

```
$ curl http://localhost:1234/api/hello?name=jim
hello, jim
```

284.6. 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

```
version: '3.8'  
services:  
  mongo:  
    ports:  
      - "27017:27017"  
  postgres:  
    ports:  
      - "5432:5432"
```

Notice how the container port# is now mapped according to how the override file has specified.

Shutdown and Start New Container Instances

```
$ unset API_PORT  
$ docker-compose down  
$ docker-compose up -d
```

Port Mappings with Compose Override File Used

	Name	State	Ports
---	---	---	---
	docker-hello-example_api_1	Up	0.0.0.0:8080->8080/tcp,:::8080->8080/tcp
	docker-hello-example_mongo_1	Up	0.0.0.0:27017->27017/tcp,:::27017->27017/tcp
	docker-hello-example_postgres_1	Up	0.0.0.0:5432->5432/tcp,:::5432->5432/tcp

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.

284.7. 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

284.8. 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

284.9. 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

```
API_PORT=9090
```

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  
API_PORT=9999  
  
$ docker-compose --env-file alt-env up -d ①  
$ docker ps  
IMAGE           PORTS  
NAMES  
dockercompose-votes-api:latest 0.0.0.0:9999->8080/tcp  
...
```

① starting network in background with an alternate environment file mapping API port to 9999

Chapter 285. Docker Compose Commands

285.1. 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 build
postgres uses an image, skipping
mongo uses an image, skipping
Building api
[+] Building 0.2s (13/13) FINISHED
=> => naming to docker.io/library/docker-hello-example:layered
```

285.2. Start Services in Foreground

We can start all the 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 up
docker-hello-example_mongo_1 is up-to-date
docker-hello-example_postgres_1 is up-to-date
Recreating docker-hello-example_api_1 ... done
Attaching to docker-hello-example_mongo_1, docker-hello-example_postgres_1, docker-
hello-example_api_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.

285.3. 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 `dockercustom-votes-it` was used in each of the running service names.

Project Name Defaults to Parent Directory Name

```
pwd  
.../svc-container/docker-hello-example  
  
$ docker-compose up  
docker-hello-example_mongo_1 is up-to-date  
docker-hello-example_postgres_1 is up-to-date  
Recreating docker-hello-example_api_1 ... done
```

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 -p foo up ①  
Creating network "foo_default" with the default driver  
Creating foo_postgres_1 ... done ②  
Creating foo_mongo_1 ... done  
Creating foo_api_1 ... done  
Attaching to foo_postgres_1, foo_mongo_1, foo_api_1
```

① manually setting project name to `foo`

② network and services all have prefix of `foo`

285.4. Start Services in Background

We can start the processes in the background by adding the `-d` option.

```
$ docker-compose up -d  
Creating network "docker-hello-example_default" with the default driver  
Creating docker-hello-example_postgres_1 ... done  
Creating docker-hello-example_mongo_1 ... done  
Creating docker-hello-example_api_1 ... done  
$ ①
```

① `-d` option starts all services in the background and returns us to our shell prompt

285.5. Access Service Logs

With the services running in the background, we can access the logs using the `docker-compose logs` command.

```
$ docker-compose logs api ①  
$ docker-compose logs -f api mongo ②  
$ docker-compose 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

285.6. 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 down
Stopping docker-hello-example_api_1    ... done
Stopping docker-hello-example_mongo_1   ... done
Stopping docker-hello-example_postgres_1 ... done
Removing docker-hello-example_api_1     ... done
Removing docker-hello-example_mongo_1   ... done
Removing docker-hello-example_postgres_1 ... done
Removing network docker-hello-example_default
```

Chapter 286. 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.herokuapp.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	6.0.0-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.

286.1. 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
```

286.2. 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
```

286.3. 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	6.0.0-SNAPSHOT	a855dabfe552	41 years ago
279MB			

Chapter 287. 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. The URLs/URIs shown in this example are consistent with what you will encounter in Heroku when deploying. However, we have much to do before then.

Where are You Going?



In the following series of lectures 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.

Assignment 4: Deployments

copyright © 2021 jim stafford (jim.stafford@jhu.edu)

This assignment contains two parts (4a-App Deploy and 4b-Docker Deploy). However, you may implement one or the other. Completing both is not a requirement. If you attempt both—please be explicit as to which one you have selected.

Both paths will result in a deployment to Heroku and an IT test against that instance. You must deploy the application using your well-known-application name and leave it deployed during the grading period. The application will **not** be deployed during a build in the grading environment.

Because of the choice of deployments, the various paths that can be taken within each option, and the fact that this assignment primarily deploys what you have already created—there is no additional support or starter modules supplied for this assignment. Everything you need should be supplied by your assignment 3 solution, the IT test for assignment 3, the deployment details covered in the Heroku and Docker lectures, and the docker-hello-example module.



Include Details Relevant to a Single Deployment Solution

Please make every attempt to follow one solution path and turn in only those details required to implement either the Spring Boot JAR or Docker deployment.

Chapter 288. Assignment 4a: Application Deployment Option

288.1. Purpose

In this portion of the assignment, you will demonstrate your knowledge of deploying a Spring Boot executable JAR to Heroku. You will:

1. create a new Heroku application with a choice of names
2. deploy a Spring Boot application to Heroku using the Heroku Maven Plugin or Git commands
3. interact with your developed application on the Internet

288.2. Overview

In this portion of the assignment you will be deploying your assignment 3 solution to Heroku as a Spring Boot JAR, making it accessible to the Internet, and be able to update with incremental changes.

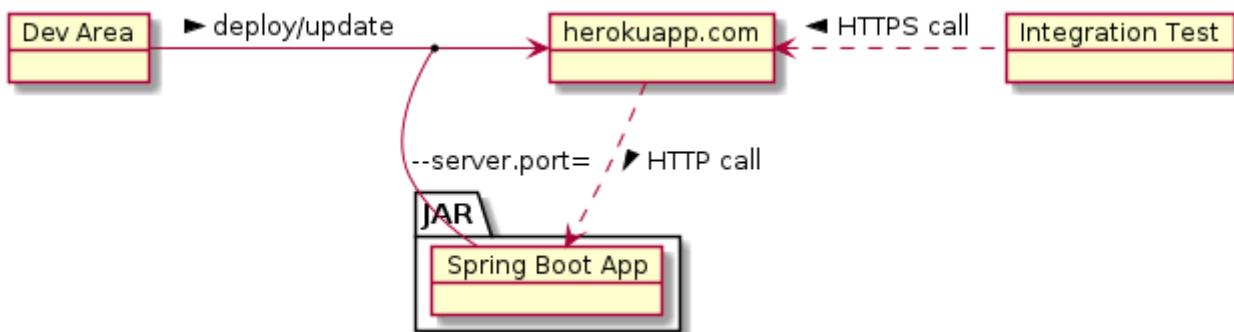


Figure 116. Spring Boot JAR Heroku Deploy

288.3. Requirements

1. Create an application name on Heroku. This may be random, a provided name, or random renamed later to a provided name.
2. Deploy your application as a Spring Boot JAR using the Heroku Maven Plugin. The profile(s) activated should use HTTP—not HTTPS added in the last assignment.
3. Provide a failsafe, IT integration test that demonstrates functionality of the deployed application on Heroku. This can be the same IT test submitted in the previous assignment adjusted to use a remote URL.
4. Turn in a source tree with complete Maven modules that will build web application. Deployment should not be a default goal in what you turn in.

288.3.1. Grading

Your solution will be evaluated on:

1. create a new Heroku application with a choice of names
 - a. whether you have provided the URL with application name of your deployed solution
2. deploy a Spring Boot application to Heroku using the Heroku Maven Plugin.
 - a. whether your solution for assignment 3 is now deployed to Heroku and functional after a normal warm-up period
3. interact with your developed application on the Internet
 - a. whether your integration test demonstrates basic application functionality in its deployed state

288.3.2. Additional Details

- Setup your Heroku account and client interface according to the course lecture and referenced Heroku reference pages.
- Your Heroku deployment and integration test can be integrated for your development purposes, but what you **turn in** must
 - assume the application is already deployed by default
 - be pre-wired with the remote Heroku URL to your application
 - be able to automatically run your IT test as part of the Maven module build.

Chapter 289. Assignment 4b: Docker Deployment Option

289.1. Docker Image

289.1.1. 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
2. make a Heroku-deployable Docker image that accepts environment variable(s)

289.1.2. Overview

In this portion of the assignment you will be building a Docker image with your Spring Boot application organized in layers

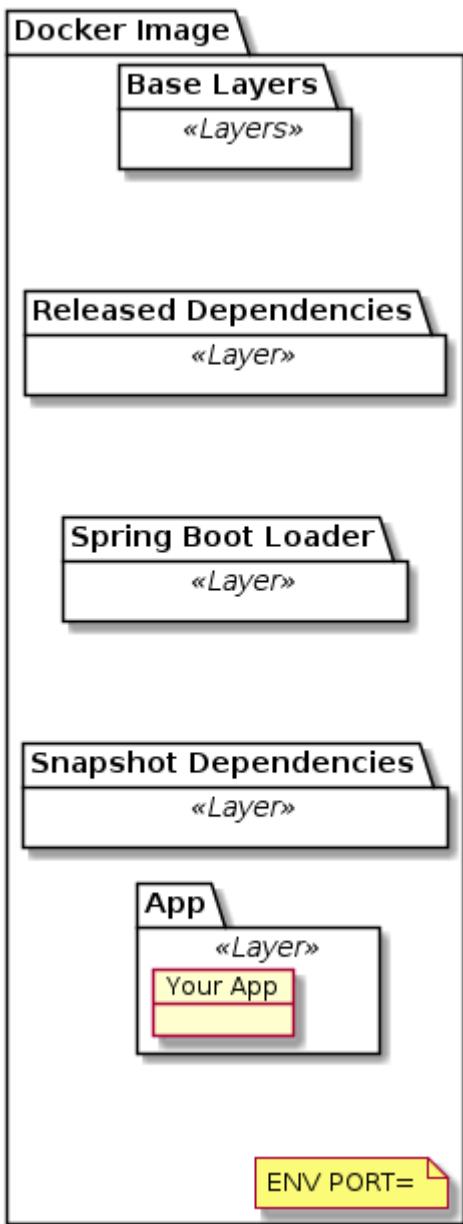


Figure 117. Layered Docker Image

289.1.3. Requirements

1. Create a layered Docker image using a Dockerfile [multi-stage build](#) that will extend a JDK image and complete the image with your Spring Boot Application broken into separate layers.
 - a. The Spring Boot application should use HTTP and not HTTPS within the container.
2. Configure the Docker image to map the `server.port` Web server property to the `PORT` environment variable when supplied.
 - a. assign a default value when not supplied
3. Turn in a source tree with complete Maven modules that will build web application.

289.1.4. 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
2. make a Heroku-deployable Docker image that accepts environment variable(s)
 - a. whether you successfully map an optional `PORT` environment variable to the `server.port` property for the Web server.

289.1.5. Additional Details

- You may optionally choose to build the Dockerfile using the [Spotify Dockerfile Maven Plugin](#)

289.2. Heroku Docker Deploy

289.2.1. Purpose

In this portion of the assignment, you will demonstrate your knowledge of provisioning a site and deploying a Docker image to Heroku. You will:

1. deploy a Docker image to Heroku

289.2.2. Overview

In this portion of the assignment you will be deploying your assignment 3 solution to Heroku as a Docker image.

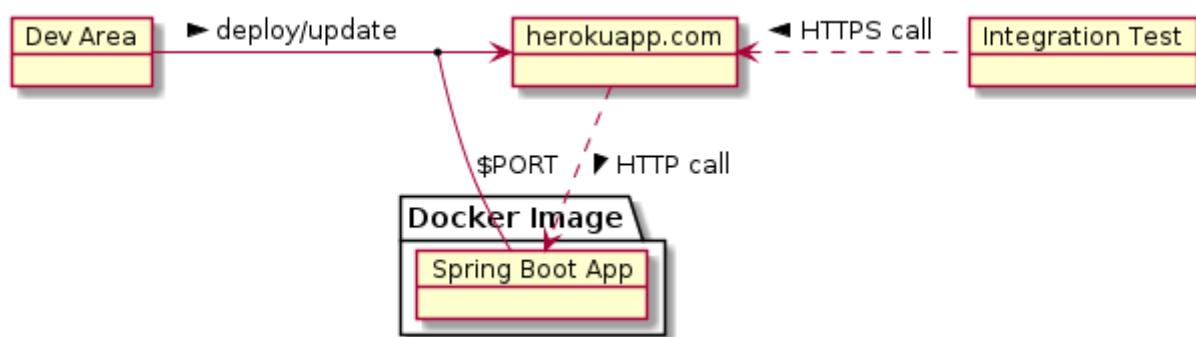


Figure 118. Spring Boot Docker Image Deploy

289.2.3. Requirements

1. Create an application name on Heroku. This may be random, a provided name, or random renamed later to a provided name.
2. Deploy your application as a Docker image using the Heroku CLI or other means. The profile(s) activated should use HTTP — not HTTPS added in the last assignment.
3. Provide an integration test that demonstrates functionality of the deployed application on Heroku. This can be the same tests submitted in the previous assignment adjusted to use a remote URL.
4. Turn in a source tree with complete Maven modules that will build web application.

Deployment should not be a default goal in what you turn in.

289.2.4. Grading

Your solution will be evaluated on:

1. deploy a Docker image to Heroku
 - a. whether you have provided the URL with application name of your deployed solution
 - b. whether your solution for assignment 3 is now deployed to Heroku, within a Docker image, and functional after a normal warm-up period
 - c. whether your integration test demonstrates basic application functionality in its deployed state

289.2.5. Additional Details

- Setup your Heroku account and client interface according to the course lecture and referenced Heroku reference pages.
- Your Heroku deployment and integration test can be integrated for your development purposes, but what you **turn in** must
 - assume the application is already deployed by default
 - be pre-wired with the remote Heroku URL to your application
 - be able to automatically run your IT test as part of the Maven module build.

RDBMS

copyright © 2021 jim stafford (jim.stafford@jhu.edu)

Chapter 290. 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.

290.1. 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

290.2. 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 291. 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.

291.1. 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**

291.2. 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

291.3. 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`

291.4. 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 web site](#). Some column types can be vendor-specific.

Table 18. 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 valuedecimal(size, digits) - fixed-point number. e.g. money
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.

291.5. 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

291.6. 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 19. Primary Key Origins

Primary Key Origins	Natural PKs	Surrogate PKs
Description	derived directly from business properties of 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.

291.7. 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. ^[56] 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 20. 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

291.8. 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 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.

291.8.1. 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.

[56] "Universally unique identifier", Wikipedia

Chapter 292. 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 293. 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.

293.1. 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.

293.2. 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 hibernate_sequence; ①
drop table IF EXISTS reposongs_song;

create sequence hibernate_sequence start 1 increment 1; 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 294. 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 "hibernate_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
```

294.1. 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.
#
```

294.2. 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 | hibernate_sequence | sequence | postgres | 8192 bytes |
 public | reposongs_song    | table    | postgres | 8192 bytes | song database
(2 rows)
```

294.3. 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 295. 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.

295.1. 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

295.2. 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
- `javax.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 javax.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

295.3. 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 296. 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.

296.1. Flyway Automated Schema Migration

"Flyway is an open-source database migration tool".^[57] 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.

296.2. Flyway Schema Source

By default, schema files^[58]

- 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
- 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
```

296.3. 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-6.0.0-SNAPSHOT.jar --spring.profiles.active=postgres  
--db.populate=false
```

296.4. 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
```

296.5. 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 hibernate_sequence;  
DROP SEQUENCE
```

296.6. 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 "hibernate_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)
```

[57] ["Flyway Documentation",Flyway Web Page](#)

[58] ["Execute Flyway Database Migrations on Startup",docs.spring.io Web Site](#)

Chapter 297. 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.

297.1. 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 21. H2 Console Windows

The screenshot shows two windows of the H2 Console. On the left is the 'Login' window, which contains 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'. Below these are 'Connect' and 'Test Connection' buttons. On the right is the 'Database browser' window, which displays a tree view of database objects. The tree includes 'REPOSONGS_SONG' with children 'ID', 'TITLE', 'RELEASED', 'ARTIST', and 'Indexes'; 'flyway_schema_history' and 'INFORMATION_SCHEMA'; 'Sequences' with 'HIBERNATE_SEQUENCE'; and 'Users' with 'H2 1.4.200 (2019-10-14)'. At the bottom of the browser window, there is a command input field containing 'call next value for hibernate_sequence; NEXT VALUE FOR PUBLIC.HIBERNATE_SEQUENCE' and a result table showing the value '1'.

297.2. Postgres CLI Access

With postgres activated, we can access the postgres server using the `psql` CLI.

```
$ docker-compose exec postgres psql -U postgres
psql (12.3)
Type "help" for help.

postgres=#
```

297.3. 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('hibernate_sequence');
nextval
-----
6
```

h2 sequence next value example

```
call next value for hibernate_sequence;
---
1
```

297.4. 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`).

297.5. 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
7 | Sledgehammer | Peter Gabriel | 1986-05-18
```

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
```

297.6. 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
```

297.7. 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
```

297.8. RDBMS Transaction

Transactions are an important and integral part of relational databases. The transactionality of a database are expressed in "ACID" properties [\[59\]](#):

- 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.

297.8.1. 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

297.8.2. 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

[59] "ACID Wikipedia Page", Wikipedia

Chapter 298. 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.^[60] 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.

298.1. 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 {
            String url = dataSource.getConnection().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 All/most JDBC commands declare throwing a `SQLException` that must be explicitly handled

298.2. 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)) {

        //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

298.3. 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, Arrays.asList(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 plans.

298.4. 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

298.5. 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.

```

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

298.6. 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();
    try(PreparedStatement call = conn.prepareStatement(sql)) {
        try (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");
            }
        }
    }
}
```

298.7. 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 hibernate_sequence"),
    POSTGRES("select nextval('hibernate_sequence')");
    private String nextvalSql;
    private Dialect(String nextvalSql) {
        this.nextvalSql = nextvalSql;
    }
    String getNextvalSql() { return nextvalSql; }
}
```

[60] "JDBC Tutorial", tutorialspoint.com

Chapter 299. 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 © 2021 jim stafford (jim.stafford@jhu.edu)

Chapter 300. 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.

300.1. 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

300.2. 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 PersistenceContext 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 301. 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 capability that would take us many chapters and weeks to cover. I will just cover the very basic setup and `@Entity` mapping at this point.

301.1. 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 that was 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
- [OpenJPA](#) from the Apache Software Foundation
- [DataNucleus](#)

301.2. 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 JPA Starter Dependencies

```
+-- org.springframework.boot:spring-boot-starter-data-jpa:jar:2.4.2:compile
|   +- jakarta.transaction:jakarta.transaction-api:jar:1.3.3:compile
|   +- jakarta.persistence:jakarta.persistence-api:jar:2.2.3:compile ①
|   +- org.hibernate:hibernate-core:jar:5.4.27.Final:compile ②
|   +- org.springframework.data:spring-data-jpa:jar:2.4.3: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

From these dependencies we have the ability to define and inject various JPA beans.

301.3. 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
@EntityScan ②
public class JPASongsApp {
```

① triggers and configures scanning for JPA Repositories

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.

301.4. 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 22. Spring Data JPA Database Connection Properties

H2 In-Memory Example Properties

```
spring.datasource.url=jdbc:h2:mem:users
```

Postgres Client Example Properties

```
spring.datasource.url=jdbc:postgresql://localhost:5432/postgres
spring.datasource.username=postgres
spring.datasource.password=secret
```

301.5. 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
```

301.6. 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 PersistenceContext. Hibernate has the additional features to allow for formatting and line termination specification.

Schema Generation to File Example

```
spring.jpa.properties javax.persistence.schema-generation.scripts.action=drop-and-create  
spring.jpa.properties javax.persistence.schema-generation.create-source=metadata  
  
spring.jpa.properties javax.persistence.schema-generation.scripts.create-target=target/generated-sources/ddl/drop_create.sql  
spring.jpa.properties javax.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` ^[61]

`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.

301.7. 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 that were mapped to the SQL substitution variables.

JPA/Hibernate SQL/JDBC Debug Properties

```
spring.jpa.show-sql=true ①  
logging.level.org.hibernate.type=trace ②
```

① prints JPA SQL commands

② prints SQL parameter values

The following cleaned up output shows the result 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.

```
Hibernate: call next value for hibernate_sequence
Hibernate: insert into reposongs_song (artist, released, title, id) values (?, ?, ?, ?)

binding parameter [1] as [VARCHAR] - [Rage Against The Machine]
binding parameter [2] as [DATE] - [2020-05-12]
binding parameter [3] as [VARCHAR] - [Recalled to Life]
binding parameter [4] as [INTEGER] - [1]
```

301.8. 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})
```

301.9. 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 class, clients can gain access to metadata and initiate a Persistence Context.

Persistance Unit/EntityManagerFactory Injection Example

```
import javax.persistence.EntityManagerFactory;  
...  
@Autowired  
private EntityManagerFactory emf;
```

301.10. JPA Persistence Context

A Persistence Context is a usage instance of a Persistence Unit and is represented by an **EntityManager**. An **@Entity** with the same identity is represented by a single instance within a Persistence Context.

Persistance Context/EntityManager Injection Example

```
import javax.persistence.EntityManager;  
...  
@Autowired  
private EntityManager em;
```

Injected **EntityManagers** 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.

[61] "JavaEE: The JavaEE Tutorial, Database Schema Creation", Oracle, JavaEE 7

Chapter 302. JPA Entity

A JPA `@Entity` is a class that is 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

```
@javax.persistence.Entity ①
@Getter
@AllArgsConstructor
@NoArgsConstructor ②
public class Song {
    @javax.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.



302.1. JPA `@Entity` Defaults

By convention and supplied annotations, the class as shown above would:

- have the entity name "Song" (important when expressing 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

302.2. JPA Overrides

Many/all of 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
- 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
...
public class Song {
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE) ②
    @Column(name = "ID") ③
    private int id;
    @Column(name="TITLE", length=255, nullable=true, insertable=true, updatable=true
)④
    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`. The default sequence name is `hibernate-sequence` for the Hibernate JPA provider.

③ re-asserting the default convention column name `ID` for the `id` field

④ re-asserting many of the default convention column mappings

Schema generation properties not used at runtime



Properties like `length` and `nullable` are only used during optional JPA schema generation and are not used at runtime.

Chapter 303. Basic JPA CRUD Commands

JPA provides an API for implementing persistence to the database through manipulation of `@Entity` instances and calls to the EntityManager.

303.1. 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;  
  
    public void create(Song song) {  
        em.persist(song);  
    }  
    ...
```

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 hibernate_sequence  
Hibernate: insert into reposongs_song (artist, released, title, id) values (?, ?, ?, ?)
```

In the managed state, any changes to the `@Entity` will result in a future `UPDATE` SQL command. 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.

303.2. 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  
    song0_.id as id1_0_0_,  
    song0_.artist as artist2_0_0_,  
    song0_.released as released3_0_0_,  
    song0_.title as title4_0_0_  
  from reposongs_song song0_  
 where song0_.id=?
```

From that point forward, the state will be returned from the Persistence Context without the need to get the state from the database.

303.3. EntityManager query

JPA provides many types of queries

- JPA Query Language (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 JPAQL 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 JPAQL 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 query
- ⑤ entity exists if row count of rows matching PK is 1. Otherwise should be 0

The following shows how our JPAQL 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(song0_.id) as col_0_0_
from reposongs_song song0_
where song0_.id=?
```

303.4. 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., prior to 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`

303.5. 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=?
```

303.6. 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 304. 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.

304.1. 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

```
javax.persistence.TransactionRequiredException: No EntityManager with actual  
transaction available for current thread - cannot reliably process 'persist' call
```

304.2. 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

304.3. 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  
} catch (Exception ex) { ②  
} finally {  
    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

304.4. 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 it 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 a transaction

304.5. 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.

```
@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 separate transaction

② **SELECT** completes in follow-on transaction

304.6. @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  
    song0_.id as id1_0_0_,  
    song0_.artist as artist2_0_0_,  
    song0_.released as released3_0_0_,  
    song0_.title as title4_0_0_  
from reposongs_song song0_  
where song0_.id=?
```

① transaction 1

② transaction 2

304.7. 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
```

304.8. 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

304.9. `@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 305. Summary

In this module we learned:

- to configure a JPA project to include project dependencies and required application properties
- to define a PersistenceContext 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 © 2021 jim stafford (jim.stafford@jhu.edu)

Chapter 306. Introduction

JDBC/SQL provided a lot of capability 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.

306.1. 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

306.2. 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 307. Spring Data JPA Repository

Spring Data JPA provides repository support for JPA-based mappings.^[62] 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.

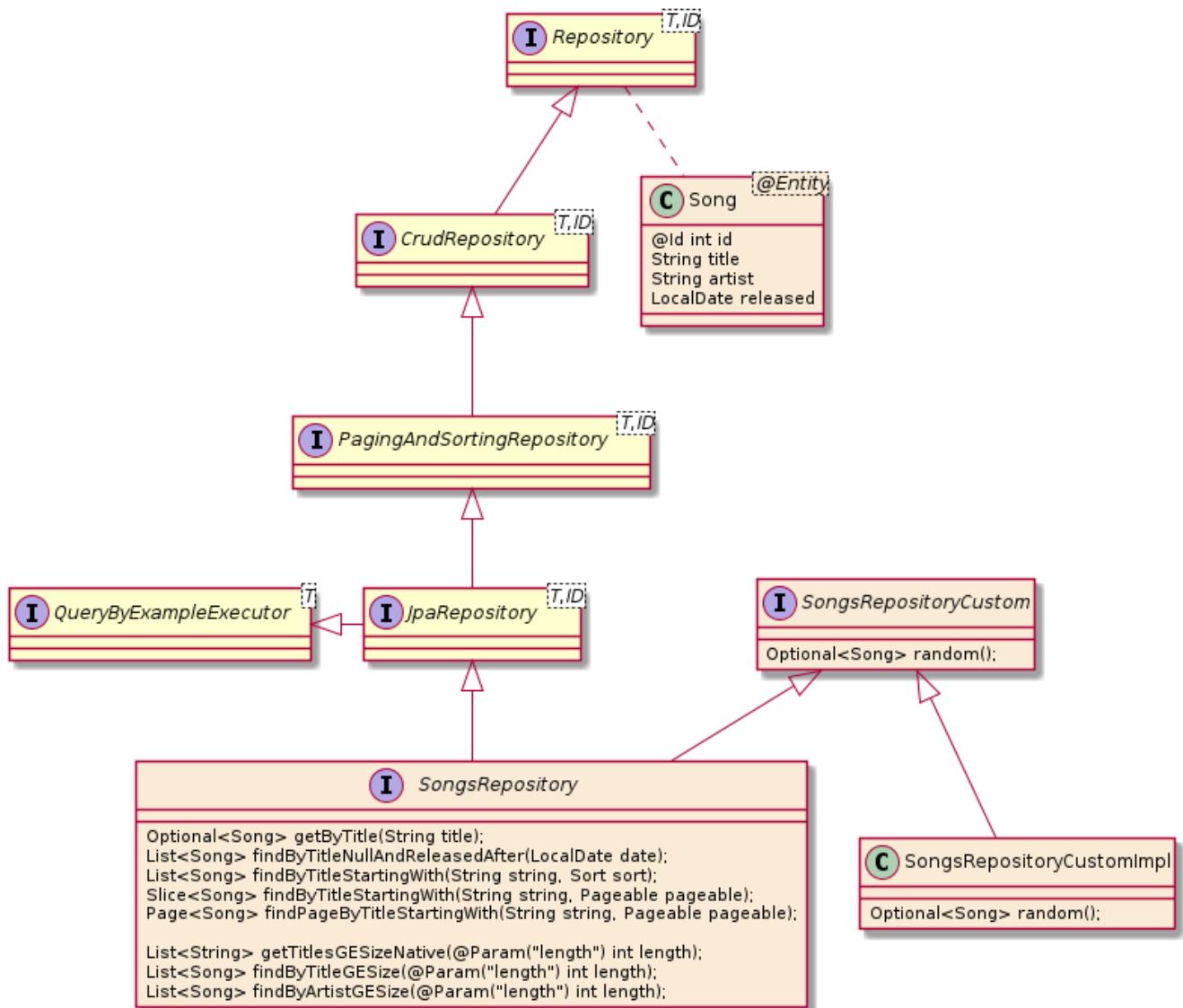


Figure 119. Spring Data JPA Repository Interfaces

[62] "Spring Data JPA - Reference Documentation"

Chapter 308. 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 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>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 309. 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.

309.1. Song @Entity

Song @Entity Example

```
@Entity  
public class Song {  
    @Id  
    private int id;
```

309.2. 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 310. Configuration

Assuming your repository and entity classes are in a package below the class annotated with `@SpringBootApplication`—all that is needed 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

310.1. 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 311. 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 var1);
    <S extends T> Iterable<S> saveAll(Iterable<S> var1);
    Optional<T> findById(ID var1);
    boolean existsById(ID var1);
    Iterable<T> findAll();
    Iterable<T> findAllById(Iterable<ID> var1);
    long count();
    void deleteById(ID var1);
    void delete(T var1);
    void deleteAll(Iterable<? extends T> var1);
    void deleteAll();
}
```

311.1. 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

```
call next value for hibernate_sequence
insert into reposongs_song (artist, released, title, id) values (?, ?, ?, ?)
```

311.2. 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 database

③ new, unmanaged `@Entity` instance has an assigned, non-default primary key value

④ object's new state is found in database

311.3. 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`.

```
select ... ①
from reposongs_song song0_
where song0_.id=?
binding parameter [1] as [INTEGER] - [1]
extracted value ([artist2_0_0_] : [VARCHAR]) - [The Beach Boys]
extracted value ([released3_0_0_] : [DATE]) - [2010-06-07]
extracted value ([title4_0_0_] : [VARCHAR]) - [If I Forget Thee Jerusalem]

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

311.4. 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();
}
```

311.5. 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(*) as col_0_0_ from reposongs_song song0_ where song0_.id=? ①
```

① `count(*)` avoids having to return all column values

311.6. 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

311.6.1. 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()).isTrue();
//when - obtaining the instance
Song dbSong = result.get();
//then - instance provided
then(dbSong).isNotNull();
```

311.6.2. 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);
```

311.7. 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();
```

311.7.1. 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 song0_ where song0_.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`.

311.7.2. 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 ... as title4_0_0_ from reposongs_song song0_ where song0_.id=? ①
```

① no `@Entity` was found/loaded as a result of this call

311.8. 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 song0_ where song0_.id=?  
delete from reposongs_song where id=?
```

deleteById will Throw Exception

 Calling `deleteById` for a non-existent `@Entity` will throw a `EmptyResultDataAccessException`.

311.9. Other CrudRepository Methods

That was a quick tour of the `CrudRepository<T, ID>` interface methods. The following snippet shows the methods not covered. Most provide convenience methods around the entire repository.

Other CrudRepository Methods

```
<S extends T> Iterable<S> saveAll(Iterable<S> var1);
Iterable<T> findAll();
Iterable<T> findAllById(Iterable<ID> var1);
long count();
void deleteAll(Iterable<? extends T> var1);
void deleteAll();
```

Chapter 312. 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. 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.

312.1. 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 song0_
order by song0_.released desc, song0_.id asc ①
```

① `Sort.by()` added the extra SQL `order by` clause

312.2. 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("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.

312.3. 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

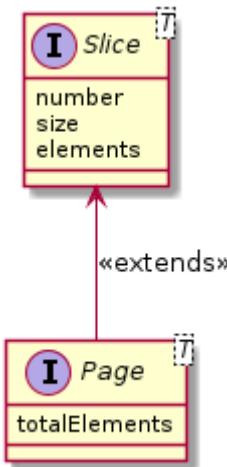


Figure 120. `Page<T>` Extends `Slice<T>`

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 not needed, use a Slice return.

312.4. 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()).isFalse();
then(songSlice.getNumber()).isEqualTo(0); ②
then(songSlice.getSize()).isEqualTo(pageSize);
then(songSlice.getNumberofElements()).isEqualTo(pageSize);

List<Song> songsList = songSlice.getContent();
then(songsList).hasSize(pageSize);
```

① `Page<T>` extends `Slice<T>`

② slice increment — first slice is 0

312.5. 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 song0_ order by song0_.released asc limit ? ①
select count(song0_.id) as col_0_0_ from reposongs_song song0_ 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)

312.6. 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();
```

312.7. 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()).isLessThanOrEqualTo(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. The **offset** parameter is added to the SQL query once we get beyond the first page,

Page Iteration SQL

```
select ... from reposongs_song song0_ order by song0_.released asc limit ? offset ? ①
select count(song0_.id) as col_0_0_ from reposongs_song song0_
```

① deeper into paging causes **offset** (in addition to **limit**) to be added to query

Chapter 313. 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> var1);  
    <S extends T> Iterable<S> findAll(Example<S> var1);  
    <S extends T> Iterable<S> findAll(Example<S> var1, Sort var2);  
    <S extends T> Page<S> findAll(Example<S> var1, Pageable var2);  
    <S extends T> long count(Example<S> var1);  
    <S extends T> boolean exists(Example<S> var1);  
}
```

313.1. Example Object

An `Example` is an interface with the ability to hold onto a probe and matcher.

313.1.1. 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

313.1.2. 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`

313.2. 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

```
//then - not found
then(foundSongs).isEmpty();
```

313.3. Primitive Types are Non-Null

The reason for the no-match is because 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 song0_
where song0_.id=0 ①
  and song0_.artist=? and song0_.title=?
order by song0_.id asc
```

① `song0_.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

313.4. 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 song0_
where song0_.id=0 or song0_.artist=? or song0_.title=? ①
order by song0_.id asc
```

① matching any ("or") of the non-null probe values

313.5. 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 song0_
where song0_.title=? and song0_.artist=? ① ②
order by song0_.id asc
```

① the primitive `int id` field is being ignored

② both `title` and `artist` fields must match

313.6. 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());
```

313.6.1. 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

```
//binding parameter [1] as [VARCHAR] - [Earth Wind and Fire]
//binding parameter [2] as [VARCHAR] - [% a God Unknown%] ①
//binding parameter [3] as [CHAR] - [\]
select ...
from reposongs_song song0_
where song0_.artist=? and (song0_.title like ? escape ?) ②
order by song0_.id asc
```

① title parameter supplied with % characters around the probe value

② title predicate uses a `LIKE`

Chapter 314. 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.

314.1. 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 in a normal manner and Spring Data JPA takes care of the implementation.

Interface Method Usage

```
//when  
Optional<Song> result = songsRepository.getByTitle(song.getTitle());  
//then  
then(result.isPresent()).isTrue();
```

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 song0_
where song0_.title=?
```

314.2. 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

314.3. Other Keywords

Other keywords include [\[63\]](#) [\[64\]](#)

- 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.

314.4. 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 song0_
where (song0_.title is null) and song0_.released>?
```

314.5. 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 song0_
where song0_.title like ? escape ?
order by song0_.id asc
```

314.6. 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 limits were placed in the query. If we had asked for a page beyond 0, an offset would have also been provided.

Slice Response Resulting SQL

```
select ...
from reposongs_song song0_
where song0_.title like ? escape ?
order by song0_.id asc limit ?
```

314.7. 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 that were not bounded by the page limits.

Page Response Resulting SQL

```
select ...
from reposongs_song song0_
where song0_.title like ? escape ?
order by song0_.id asc
limit ? ①
select count(song0_.id) as col_0_0_
from reposongs_song song0_
where song0_.title like ? escape ? ②
```

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

[63] "*Query Creation*", Spring Data JPA - Reference Documentation

[64] "*Derived Query Methods in Spring Data JPA*", Atta

Chapter 315. 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();
```

315.1. 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 object's 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 song0_
where length(song0_.artist)>=?
```

Chapter 316. @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 song0_
where length(song0_.title)>=?
```

316.1. @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

Chapter 317. 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 PagingAndSortingRepository<T, ID>,  
QueryByExampleExecutor<T> {  
    void flush();  
    <S extends T> S saveAndFlush(S entity);  
  
    void deleteInBatch(Iterable<T> entities);  
    void deleteAllInBatch();  
  
    T getOne(ID id);
```

317.1. 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 CrudRepository<T, ID> extends Repository<T, ID> {  
    Iterable<T> findAll();  
    ...
```

Concrete Spring Data JPA Extensions

```
public interface JpaRepository<T, ID> extends PagingAndSortingRepository<T, ID>,  
QueryByExampleExecutor<T> {  
    @Override  
    List<T> findAll(); ①  
    ...
```

① `List<T>` extends `Iterable<T>`

317.2. 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 prior to 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: call next value for hibernate_sequence  
Hibernate: call next value for hibernate_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()` prior to executing.

317.3. 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

317.4. 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.

317.4.1. 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 thru Reference within Active Transaction

```
@Test  
@Transactional  
void ref_session() {  
    ...  
    //when - obtaining a reference with a session  
    Song dbSongRef = songsRepository.getOne(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 database
- ② still only dealing with the unresolved reference up and to this point
- ③ actual object resolved from database at this point

317.4.2. 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 thru Reference Outside of Transaction

```
import org.hibernate.LazyInitializationException;  
...  
@Test  
void ref_no_session() {  
    ...  
    //when - obtaining a reference without a session  
    Song dbSongRef = songsRepository.getOne(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

317.4.3. Bogus Reference

The following example shows that the reference is never attempted to be resolved until something

is needed from the object it represents — beyond its primary key.

Reference Never Resolved until Demand

```
import javax.persistence.EntityNotFoundException;  
...  
@Test  
@Transactional  
void ref_not_exist() {  
    //given  
    int doesNotExist=1234;  
    //when  
    Song dbSongRef = songsRepository.getOne(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 318. Custom Queries

Sooner or later, a repository action requires some complexity that is 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.

318.1. 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();  
}
```

318.2. 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

318.3. 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".

```
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`

318.4. 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

318.5. 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 in order to make these calls:

- SongsRepository
- EntityManager

318.6. 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.

318.7. 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.JpaRepository;
...
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

318.8. 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(song0_.id) as col_0_0_
  from reposongs_song song0_
select ...
  from reposongs_song song0_
 limit ? offset ?
```

Chapter 319. 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
- 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

319.1. 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 orm.xml source file that would allow for query changes without recompiling

JPA Repository End-to-End Application

copyright © 2021 jim stafford (jim.stafford@jhu.edu)

Chapter 320. 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, marshaling/de-marshaling using Jackson and JAXB, web APIs/controllers, and clients. This lecture will put them all together.

320.1. 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

320.2. Objectives

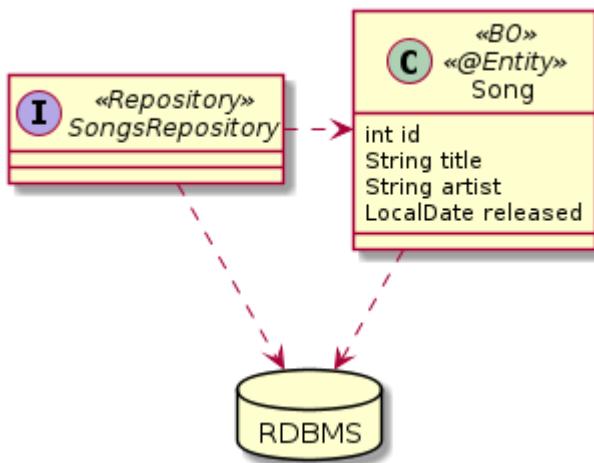
At the conclusion of this lecture and related exercises, the student will be able to:

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 321. BO/DTO Component Architecture

321.1. 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 121. BO Class Mapped to DB as JPA `@Entity`

The primary focus of our BO classes is to map business implementation concepts to the database.

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  
...  
public class Song {  
    @Id @GeneratedValue(strategy = GenerationType.SEQUENCE)  
    private int id;  
    ...
```

321.2. 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.

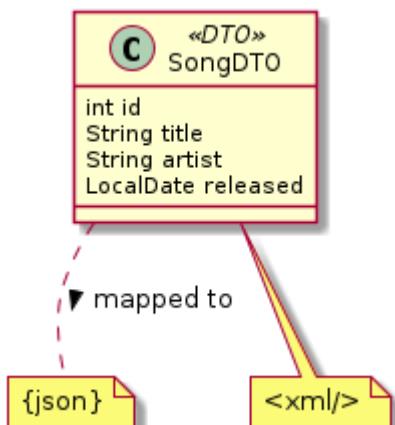


Figure 122. DTO

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

```
@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;
```

① Jackson JSON requires very little to no annotations for simple mappings

② XML mappings require more detailed definition to be complete

321.3. 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

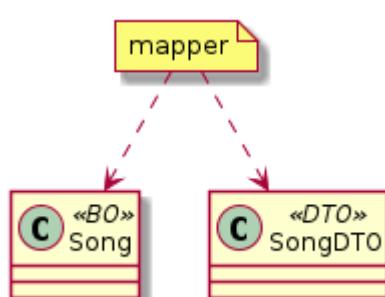


Figure 123. BO to DTO Mapping

We have several options on how to organize this role.

321.3.1. 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.

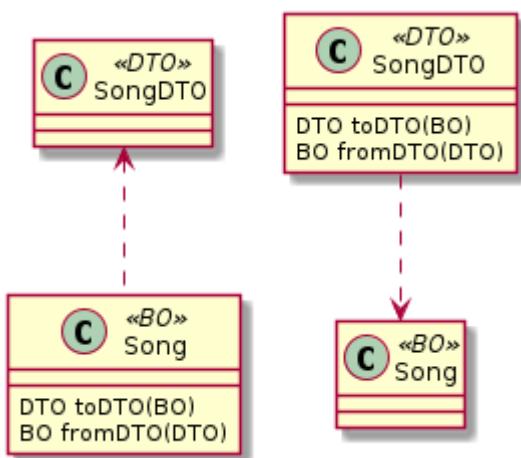


Figure 124. BO to DTO Self Mapping

321.3.2. 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.

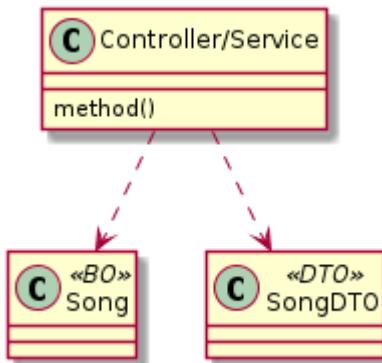


Figure 125. BO to DTO Method Self Mapping

321.3.3. 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

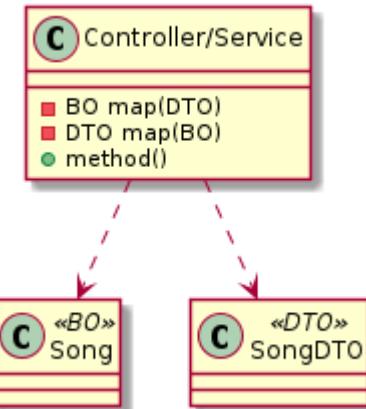


Figure 126. BO/DTO Helper Method Mapping

321.3.4. 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.

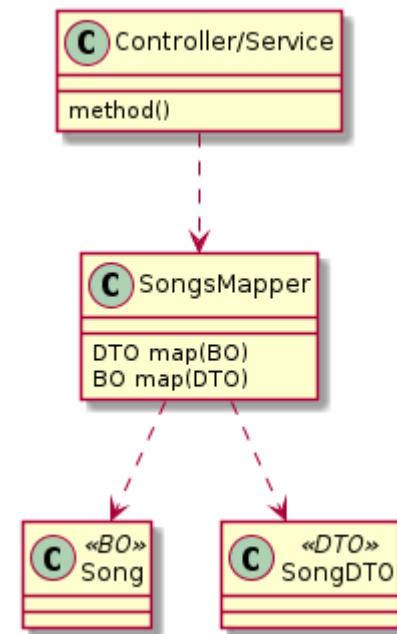


Figure 127. BO/DTO Helper Class Mapping

321.3.5. 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](#))^[65]^[66]
 - Benefit: declarative language and inferred DIY mapping options
 - Drawbacks:
 - relies on reflection and other generalizations for mapping which add to overhead

- non-trivial mappings can be complex to understand

[65] "*Performance of Java Mapping Frameworks*", Baeldung

[66] "*any tool for java object to object mapping?*", Stack Overflow

Chapter 322. Service Architecture

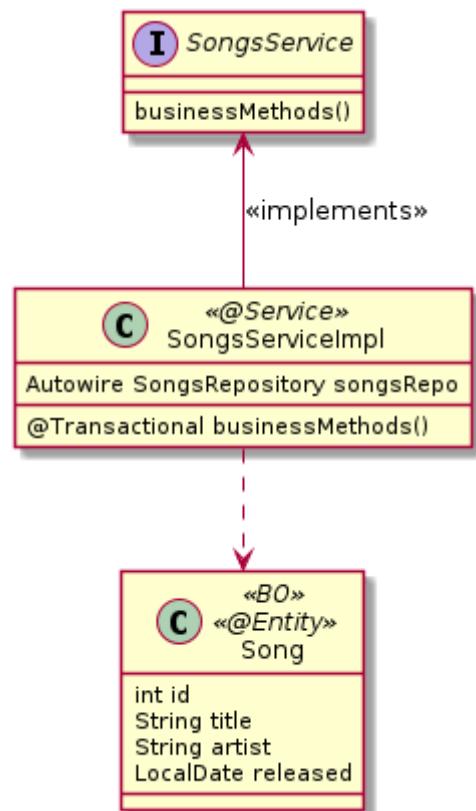
Services—with the aid of BOs—implement the meat of the business logic.

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;  
    ...
```



322.1. 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.

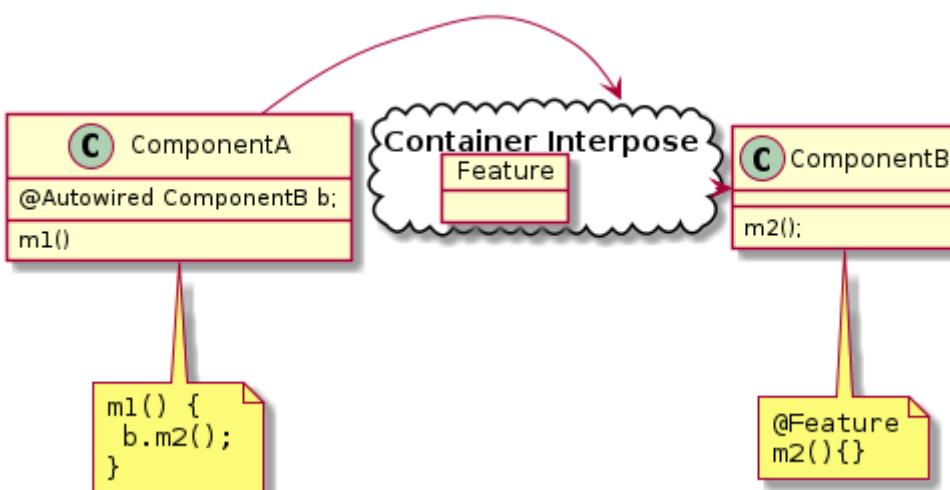


Figure 128. Container Interpose

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.

322.1.1. Buddy Method Boundary

The methods within a component class are not 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.

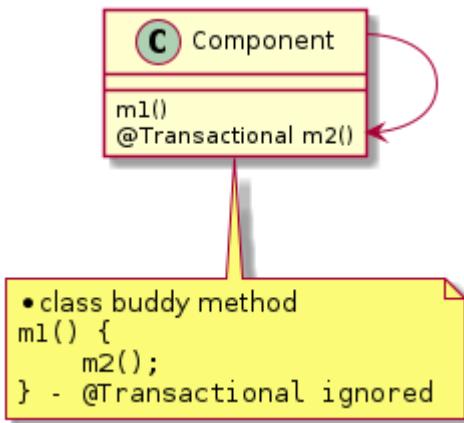


Figure 129. Buddy Method Boundary

322.1.2. 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.

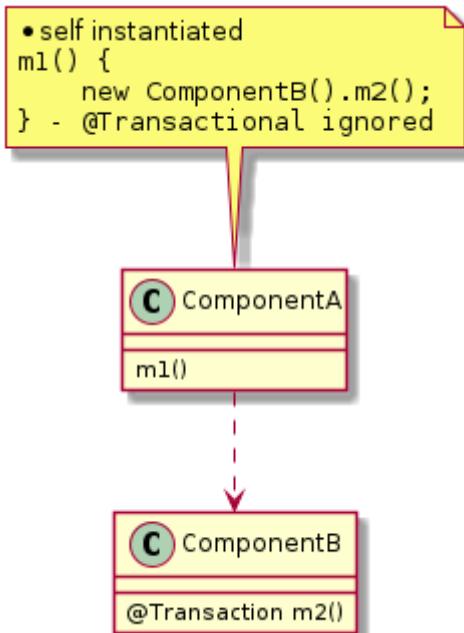


Figure 130. Self Instantiated Method Boundary

322.1.3. 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.

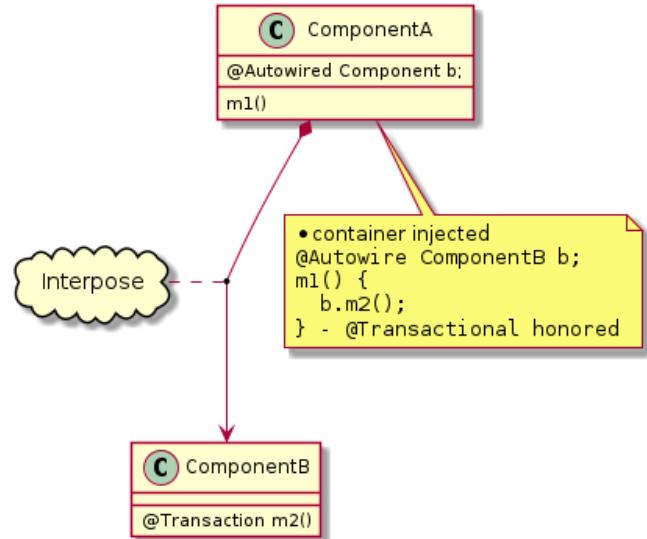


Figure 131. Container Injected Method Boundary

322.2. 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.

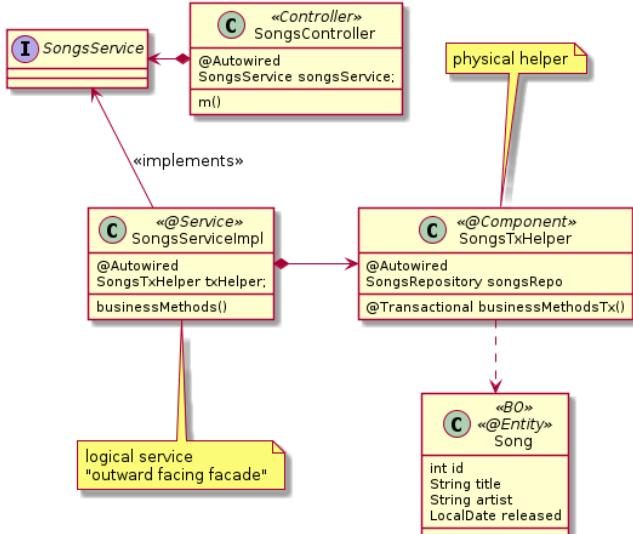


Figure 132. Single Service Expressed as Multiple Components

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`.

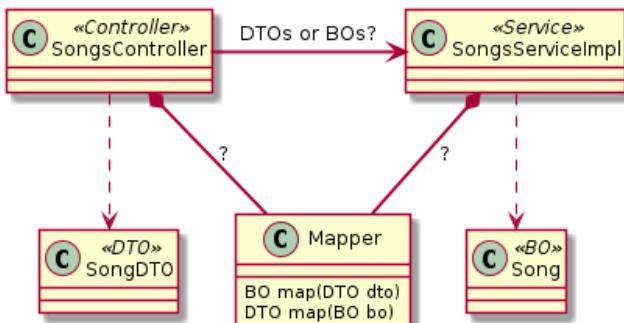
Conceptual Services may be broken into Multiple Physical Components

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 323. 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 133. BO/DTO Interface Decisions

323.1. 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 `@Controller` 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.

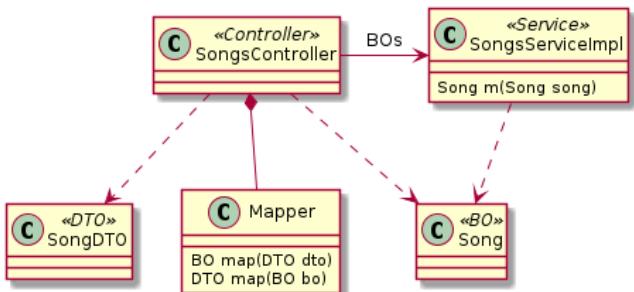


Figure 134. API Maps DTO to BO for Service Interface

323.2. @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 `@Controller` and all `@Service` clients work with DTOs.

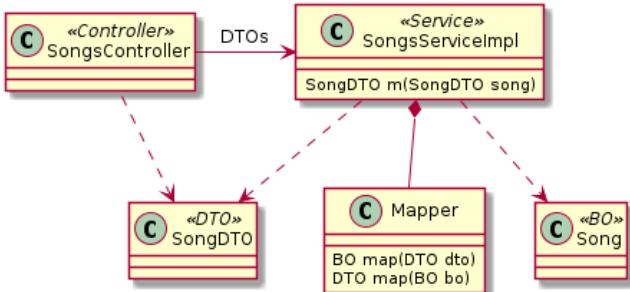
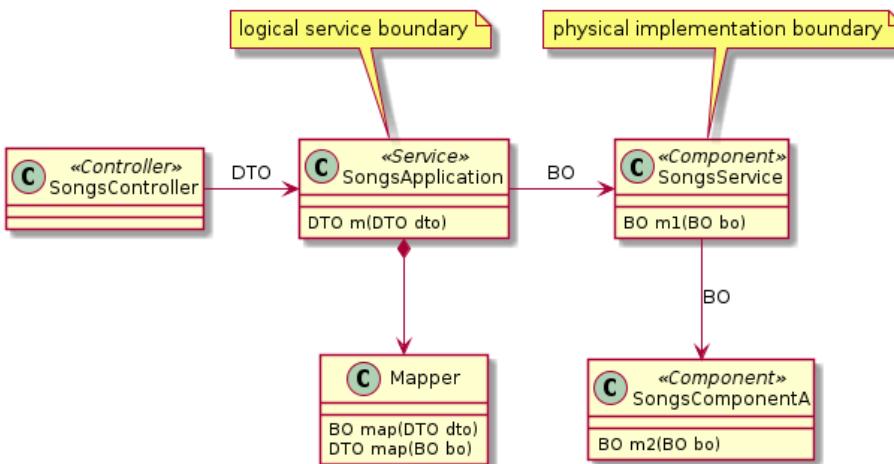


Figure 135. Service Maps DTO in Service Interface to BO

- 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.

323.3. 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 324. Implementation Details

With architectural decisions understood, lets take a look at some of the key details of the end-to-end application.

324.1. 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  
...  
public class Song {  
    @Id @GeneratedValue(strategy = GenerationType.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;  
}
```

324.2. 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.

```
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

324.2.1. 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)
private LocalDate released;

public static class LocalDateJaxbAdapter extends XmlAdapter<String, LocalDate> {
    @Override
    public LocalDate unmarshal(String text) {
        return LocalDate.parse(text, DateTimeFormatter.ISO_LOCAL_DATE);
    }
    @Override
    public String marshal(LocalDate timestamp) {
        return DateTimeFormatter.ISO_LOCAL_DATE.format(timestamp);
    }
}
```

324.3. 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"
}
```

324.4. 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>
```

```
<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>
```

324.5. 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. The DTO classes have a direct dependency on their non-DTO siblings.
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.
3. Although technically I could use either, I chose to use the Spring Data types in the `@Service` interface when expressing paging and performed the Spring Data to DTO mapping 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.

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.

324.5.1. 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.

324.5.2. 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

324.5.3. 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

324.5.4. 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"/>
```

324.6. 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.

324.6.1. 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

324.6.2. 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

324.6.3. 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

324.6.4. 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

```
SongsPageDTO pageDTO = request.exchange()  
    .expectStatus().isOk()  
    .returnResult(SongsPageDTO.class)  
    .getResponseBody().blockFirst();  
Page<SongDTO> page = pageDTO.toPage(); ①  
Pageable pageable = ... 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 325. 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.

325.1. 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;
    }
    ...
}
```

325.2. 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 326. Service Tier

The SongsService `@Service` encapsulates the implementation of our management of Songs.

326.1. 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

326.2. 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.

326.3. 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

326.4. 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 327. 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.

327.1. 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

327.2. 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

327.3. 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 `@XmlRootElement` is provided.

Chapter 328. 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 © 2021 jim stafford (jim.stafford@jhu.edu)

Chapter 329. Introduction

This lecture will introduce working with MongoDB database using the Mongo shell.

329.1. 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

329.2. 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 330. Mongo Concepts

Mongo is a document-oriented database. This type of database enforces very few rules when it comes schema. About the only rules that exists 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

330.1. Mongo Terms

The table below lists a few keys terms associated with MongoDB.

Table 23. 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>

330.2. 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 }  
    { "field53": value513, "field53": value513, "field54": value514 }  
  ]  
}
```

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 331. 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.

331.1. 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 `$EJAVA_EXAMPLES_ROOT/env` directory.

Starting Docker-Compose MongoDB

```
$ cd $EJAVA_EXAMPLES_ROOT/env
$ 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 ejava_mongodb_1 ①
```

① image is running with name ejava_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`.

331.2. 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`

331.3. 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`

331.4. 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.

331.5. 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 332. Basic CRUD Commands

332.1. 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
```

332.2. 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") }
```

332.3. 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

332.4. 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
}
```

332.5. 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

332.6. 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

332.7. 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") }
```

332.8. 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")  
}
```

332.9. 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

332.10. 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")
}
```

332.11. 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 333. 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.

333.1. 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" }
```

333.2. `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

333.3. 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 334. 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.

334.1. 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.

334.2. 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)

334.3. 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" }
```

334.4. 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}}} ])
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 335. Helpful Commands

This section contains a set if helpful Mongo shell commands. It will be populated over time.

335.1. 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
```

335.2. 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&gssapiServiceName=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" : "Dorcus 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 336. 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 © 2021 jim stafford (jim.stafford@jhu.edu)

Chapter 337. 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

is a repository-based API from Spring Data that is consistent with Spring Data JPA

This lecture covers implementing interactions with a 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.

337.1. Goals

The student will learn:

- to setup 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

337.2. 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 an (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 338. Mongo Project

Except for the possibility of indexes and defining specialized collection features—there is not the same schema rigor required to bootstrap a Mongo 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 Mongo shell during development

338.1. Mongo Project Dependencies

The following snippet shows a dependency declaration for MongoDB APIs.

Mongo 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 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:2.4.2:compile
[INFO] |   +- org.mongodb:mongodb-driver-sync:jar:4.1.1:compile
[INFO] |   |   +- org.mongodb:bson:jar:4.1.1:compile
[INFO] |   |   \- org.mongodb:mongodb-driver-core:jar:4.1.1:compile
[INFO] |   \- org.springframework.data:spring-data-mongodb:jar:3.1.3:compile
[INFO] |       +- org.springframework:spring-tx:jar:5.3.3:compile
[INFO] |       \- org.springframework.data:spring-data-commons:jar:2.4.3: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.

338.2. Mongo Project Integration Testing Options

MongoDB is written in C++. That means that we cannot simply instantiate MongoDB within our integration test JVM. We have at least three options:

- [Fongo](#) in-memory MongoDB implementation

- Flapdoodle embedded MongoDB and Auto Configuration or referenced Maven plugins:
 - maven-mongodb-plugin
 - embedmongo-maven-plugin
- Testcontainers Docker wrapper

Each should be able to do the job for what we want to do here. However,

- Although Fongo is an in-memory solution, it is not MongoDB and edge cases may not work the same as a real MongoDB instance.
- 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 Documentation](#) discusses Flapdoodle and the [Spring Boot Embedded Mongo AutoConfiguration](#) seamlessly integrates Flapdoodle with a few options. 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. ^[67]

338.3. Flapdoodle Test Dependencies

This lecture will use the Flapdoodle Embedded Mongo 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</artifactId>
  <scope>test</scope>
</dependency>
```

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 that is embedded about this capability is the logical management feel. The library downloads a MongoDB instance (cached), starts, and stops the instance as part of running the test. Spring Data MongoDB includes a starter that 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.

338.4. MongoDB Access Objects

There are two primary beans of interest when we connect and interact with MongoDB: MongoClient and MongoOperations/MongoTemplate.

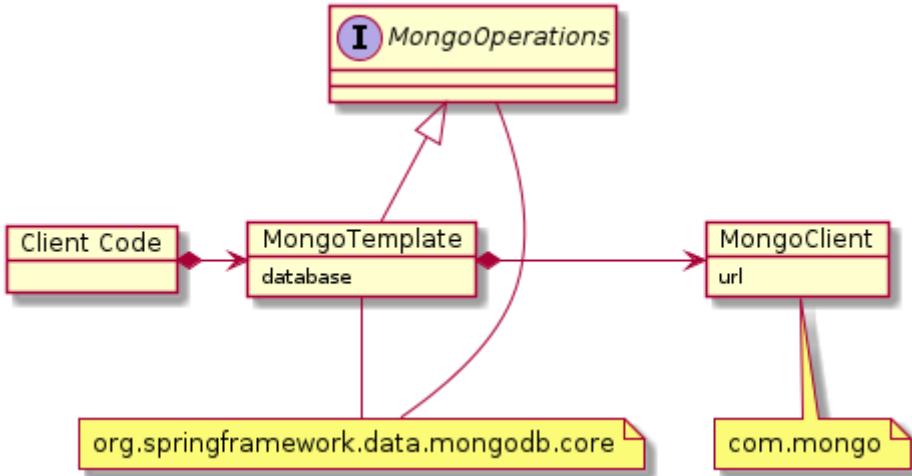


Figure 136. Injectable MongoDB Access Objects

- **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 Spring Boot Embedded MongoDB Auto Configuration class will instantiate a MongoDB instance using Flapdoodle and instantiate a `MongoClient` that references that instance.

338.5. 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.

`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

```

338.6. 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`

338.7. Disabling Embedded MongoDB

By default, Spring Boot will automatically use the Embedded MongoDB and Flapdoodle test instance for our MongoDB. For development, we may want to work against a live MongoDB instance so that 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=\  
org.springframework.boot.autoconfigure.mongo.embedded.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

```
@Configuration  
@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`

338.8. @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

What we can do is 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. 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

338.9. TestProfileResolver

I implemented a simple class based on an example from the internet from Amit Kumar.^[68] 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

```
//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 DefaultActiveProfilesResolver defaultResolver = new
DefaultActiveProfilesResolver();

    @Override
    public String[] resolve(Class<?> testClass) {
        return System.getProperties().containsKey(PROFILE_KEY) ?
            //return profiles expressed in property as array of strings
            System.getProperty(PROFILE_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

338.10. 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

338.11. 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.

[67] *"Fast and stable MongoDB-based tests in Spring"*, Piotr Kubowicz, Dec 2020

[68] *"Overriding Active Profiles in Spring Boot Integration Tests"*, Amit Kumar, 2018

Chapter 339. 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.

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  
@Builder  
@With  
@AllArgsConstructor  
public class Book {  
    @Id  
    private String id;  
    @Setter  
    @Field(name="title")  
    private String title;  
    @Setter  
    private String author;  
    @Setter  
    private LocalDate published;  
}
```

339.1. Property Mapping

339.1.1. 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 "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.

339.1.2. Primary Key Mapping

The MongoDB `_id` field will be mapped to a field that either

- is called `id`
- is annotated with `@Id`
- is mapped to field `_id` using `@Field` annotation

Primary Key Mapping

```
import org.springframework.data.annotation.Id;  
  
① @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.

339.2. 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;  
  
① @Field(name="title") ①  
private String titleXYZ;
```

① maps Java property `titleXYZ` to MongoDB document field `title`

We can annotated a property with `@Transient` to prevent a property from being stored in the database.

```
import org.springframework.data.annotation.Transient;  
  
① @Transient  
private String dontStoreMe;
```

① `@Transient` excludes the Java property from being mapped to the database

339.3. 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-arg constructor)

Given our example, the default constructor will be used.

339.4. 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 340. Command Types

MongoTemplate offers different types of command interactions

Whole Document

complete document passed in as argument and/or returned as 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 341. 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.

341.1. 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 following [example](#).

341.1.1. 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

341.1.2. 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

341.2. 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.

341.2.1. 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());
```

341.2.2. 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.id, Book.class);
then(dbBook.getTitle()).isEqualTo(updatedBook.getTitle());
then(dbBook.getAuthor()).isEqualTo(updatedBook.getAuthor());
then(dbBook.getPublished()).isEqualTo(updatedBook.getPublished());
```

341.3. 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 342. 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 next.

342.1. `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.id, 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

No document found 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(id, Book.class);
//then
then(dbBook).isNull();
```

Chapter 343. 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 field and explicit mapping type when supplying collection name in full command

343.1. 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"}}
```

343.2. 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 344. 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 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)

344.1. 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

344.2. 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 345. Paging

In conjunction with `find` commands, we need to soon look to add paging instructions in order to sort and slice up the results into page-sized bites. `RestTemplate` offers two primary ways to express paging

- Query configuration
- Pagable command parameter

345.1. 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<Book> 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);
```

345.2. 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.

345.3. 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 346. 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 a `author` field and match a regular expression.

Example Aggregation Pipeline Call

```
//given
int minLength = ...
Set<String> ids = savedBooks.stream() ... //return IDs od 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 347. 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

347.1. 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 accomplish this. ^[69]



MongoDB Multi-Document Transactions and not the Normal Path

Just because you can implement multi-document atomic transactions and references between documents, don't use 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.

347.2. 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.

347.3. Isolation

Within the context of a single document change — MongoDB ^[70]

- 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

347.4. Durability

The durability of a Mongo 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.

[69] "*Spring Data MongoDB Transactions*", baeldung, 2020

[70] "*Read Isolation, Consistency, and Recency*", MongoDB Manual, Version 4.4

Chapter 348. 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 © 2021 jim stafford (jim.stafford@jhu.edu)

Chapter 349. 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. However, 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.

349.1. 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

349.2. 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 350. Spring Data MongoDB Repository

Spring Data MongoDB provides repository support for `@Document`-based mappings.^[71] 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.

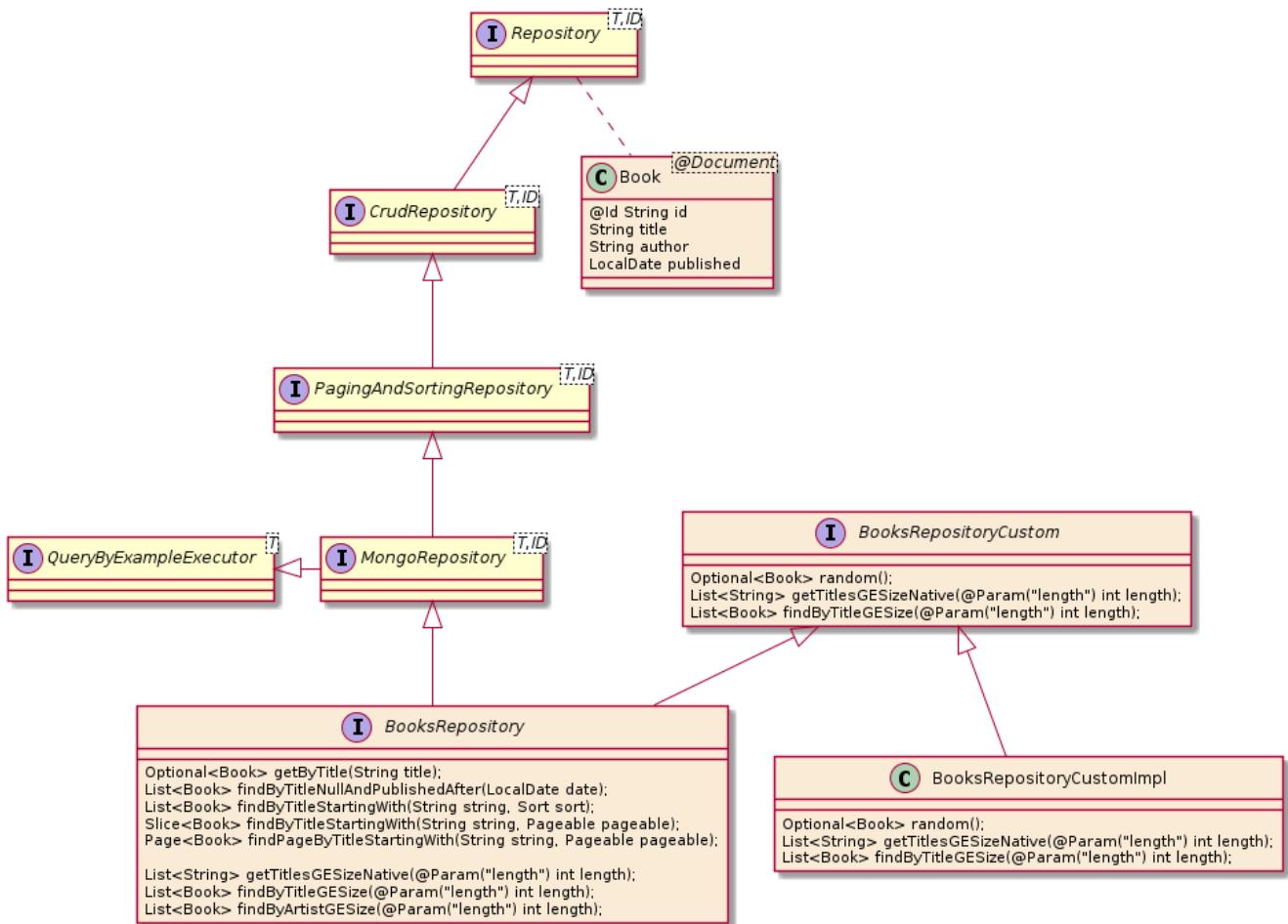


Figure 137. Spring Data MongoDB Repository Interfaces

[71] "Spring Data MongoDB - Reference Documentation"

Chapter 351. 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`.

`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 Mongo.

`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 352. BooksRepository

All we need to create a functional repository is a `@Document` class and a primary key type. 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.

352.1. Book `@Document`

Book @Document Example

```
@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;
```

352.2. 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 353. 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

353.1. 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 354. 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 var1);  
    <S extends T> Iterable<S> saveAll(Iterable<S> var1);  
    Optional<T> findById(ID var1);  
    boolean existsById(ID var1);  
    Iterable<T> findAll();  
    Iterable<T> findAllById(Iterable<ID> var1);  
    long count();  
    void deleteById(ID var1);  
    void delete(T var1);  
    void deleteAll(Iterable<? extends T> var1);  
    void deleteAll();  
}
```

354.1. 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

354.2. 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

354.3. CrudRepository save()/Update Resulting Mongo Command

Watching the low-level Mongo commands, we can see that Mongo's built-in `upsert` capability allows the client to perform the action without a separate query.

Mongo Update Command Performed with Upsert

```
update{"q":{"_id":{"$oid":"606cbfc0932e084392422bb6"}},  
      "u":{"_id":{"$oid":"606cbfc0932e084392422bb6"},"title":"new title","author":...},  
      "multi":false,  
      "upsert":true}
```

354.4. 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 Mongo 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

354.5. 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

354.5.1. 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());
```

354.5.2. 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);
```

354.6. 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();
```

354.6.1. 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);
```

354.7. 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());
```

354.8. 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

```
<S extends T> Iterable<S> saveAll(Iterable<S> var1);  
Iterable<T> findAll();  
Iterable<T> findAllById(Iterable<ID> var1);  
long count();  
void deleteAll(Iterable<? extends T> var1);  
void deleteAll();
```

Chapter 355. 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.

355.1. 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 Mongo command was impacted by the `Sort.by()` parameter.

Sort.by() Example Mongo Command

```
query: {}
sort: { published: -1, _id: 1 } ①
projection: {}"
```

① `Sort.by()` added the extra sort parameters to Mongo command

355.2. 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.

355.3. 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

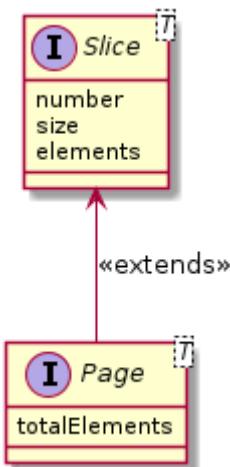


Figure 138. `Page<T>` Extends `Slice<T>`

355.4. 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);
then(bookSlice.getNumberofElements()).isEqualTo(pageSize);

List<Book> booksList = bookSlice.getContent();
then(booksList).hasSize(pageSize);
  
```

① `Page<T>` extends `Slice<T>`

② slice increment—first slice is 0

355.5. 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());
```

355.6. 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();
```

355.7. 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()).isLessThanOrEqual(pageSize);
    then(bookSlice.getNumberElements()).isLessThanOrEqual(pageSize);
    then(((Page)bookSlice).getTotalElements()).isEqualTo(savedBooks.size());//unique
    to Page
}
then(bookSlice.hasNext()).isFalse();
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 356. 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> var1);  
    <S extends T> Iterable<S> findAll(Example<S> var1);  
    <S extends T> Iterable<S> findAll(Example<S> var1, Sort var2);  
    <S extends T> Page<S> findAll(Example<S> var1, Pageable var2);  
    <S extends T> long count(Example<S> var1);  
    <S extends T> boolean exists(Example<S> var1);  
}
```

356.1. Example Object

An `Example` is an interface with the ability to hold onto a probe and matcher.

356.1.1. 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

356.1.2. 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`

356.2. 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.

356.3. 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

356.4. 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());
```

356.4.1. 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 357. 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.

357.1. 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.

357.2. Query Keywords

Spring Data MongoDB 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

357.3. Other Keywords

Other keywords are clearly documented in the JPA reference [\[72\]](#) [\[73\]](#)

- 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.

357.4. 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);
```

357.5. 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);
```

357.6. 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

357.7. 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

[72] "Query Creation", Spring Data JPA - Reference Documentation

[73] "Derived Query Methods in Spring Data JPA", Atta

Chapter 358. @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

358.1. @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 24. @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	

Chapter 359. 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 entity); ①
<S extends T> List<S> insert(Iterable<S> entities);

<S extends T> List<S> saveAll(Iterable<S> entities); ②
List<T> findAll();
List<T> findAll(Sort sort);
<S extends T> List<S> findAll(Example<S> example);
<S extends T> List<S> findAll(Example<S> example, Sort sort);
```

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

Chapter 360. 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.

360.1. 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();  
}
```

360.2. 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

360.3. 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".

```
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, 1); 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`

360.4. 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

360.5. 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

360.6. 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

360.7. 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

360.8. 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();
```

360.9. 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 361. 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 © 2021 jim stafford (jim.stafford@jhu.edu)

Chapter 362. 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, marshaling/de-marshaling 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. However, 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.

362.1. 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

362.2. Objectives

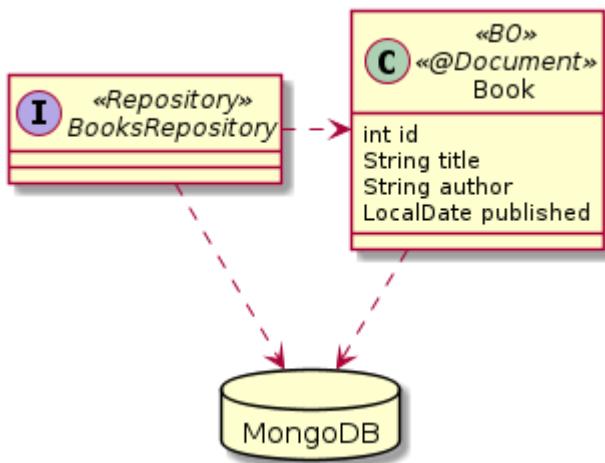
At the conclusion of this lecture and related exercises, the student will be able to:

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 363. BO/DTO Component Architecture

363.1. 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 139. 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.

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;
}
```

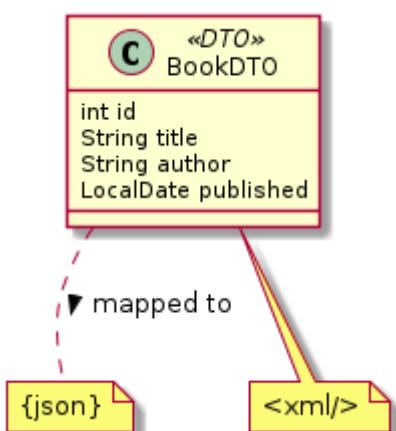
① `@Document.collection` used to define the DB collection to use — otherwise uses name of class

- 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

363.2. 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 140. DTO

The primary focus of our DTO classes is to map business interface concepts to a portable exchange format.

363.3. 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.

```
@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

363.4. 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

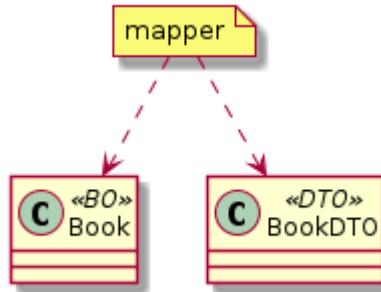


Figure 141. BO to DTO Mapping

We have several options on how to organize this role.

363.4.1. 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.

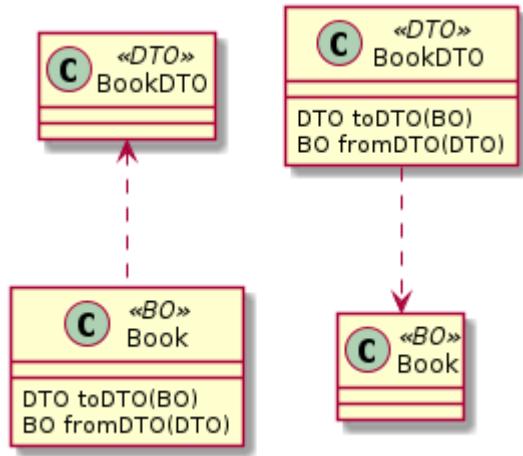


Figure 142. BO to DTO Self Mapping

363.4.2. 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.

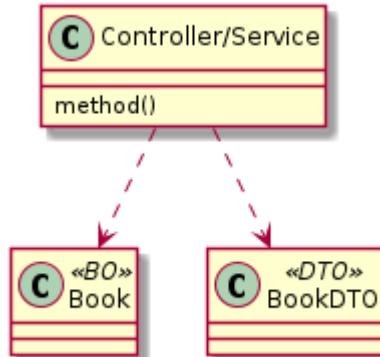


Figure 143. BO to DTO Method Self Mapping

363.4.3. 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

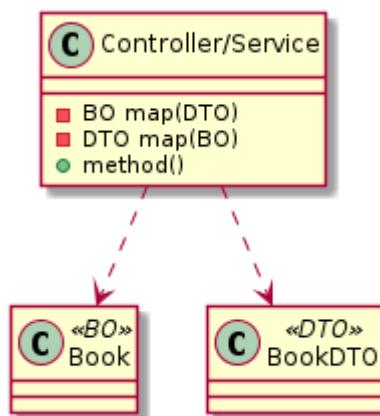


Figure 144. BO/DTO Helper Method Mapping

363.4.4. 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.

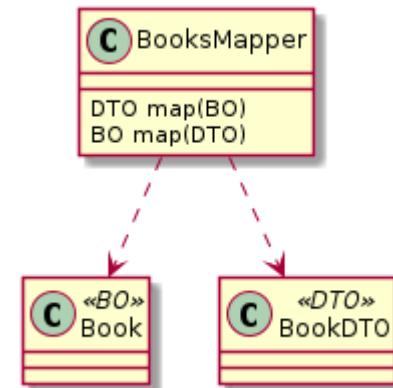


Figure 145. BO/DTO Helper Class Mapping

363.4.5. 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](#)) ^[74] ^[75]
 - Benefit: declarative language and inferred DIY mapping options
 - Drawbacks:
 - relies on reflection and other generalizations for mapping which add to overhead
 - non-trivial mappings can be complex to understand

[74] "Performance of Java Mapping Frameworks", Baeldung

[75] "any tool for java object to object mapping?", Stack Overflow

Chapter 364. Service Architecture

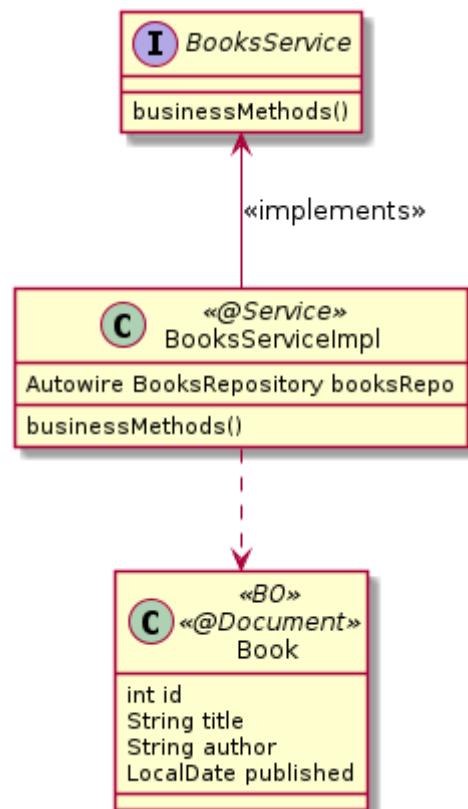
Services—with the aid of BOs—implement the meat of the business logic.

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;  
    ...
```



364.1. 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.

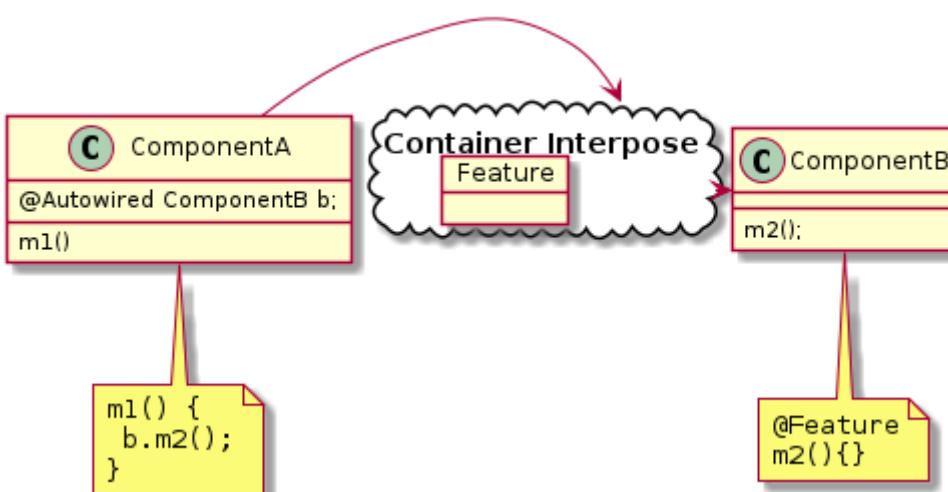


Figure 146. Container Interpose

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.

364.1.1. Buddy Method Boundary

The methods within a component class are not 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.

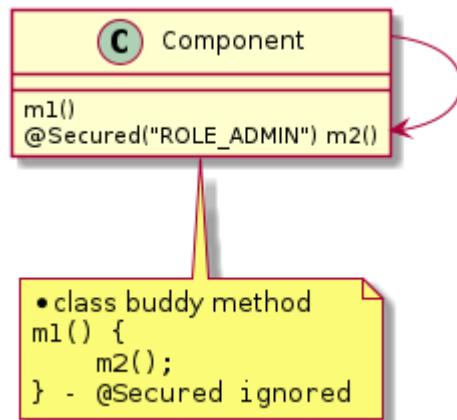


Figure 147. Buddy Method Boundary

364.1.2. 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.

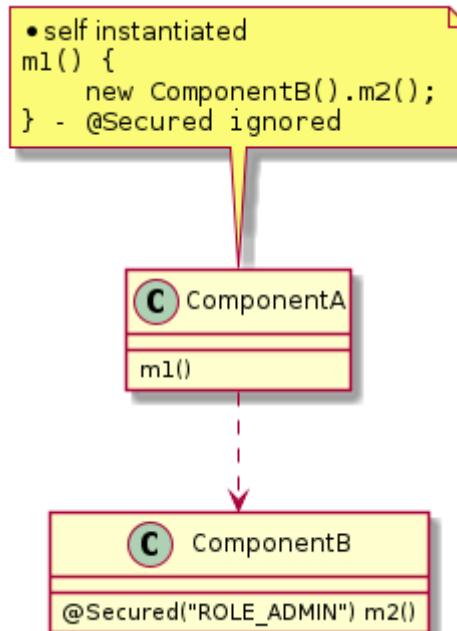


Figure 148. Self Instantiated Method Boundary

364.1.3. 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.

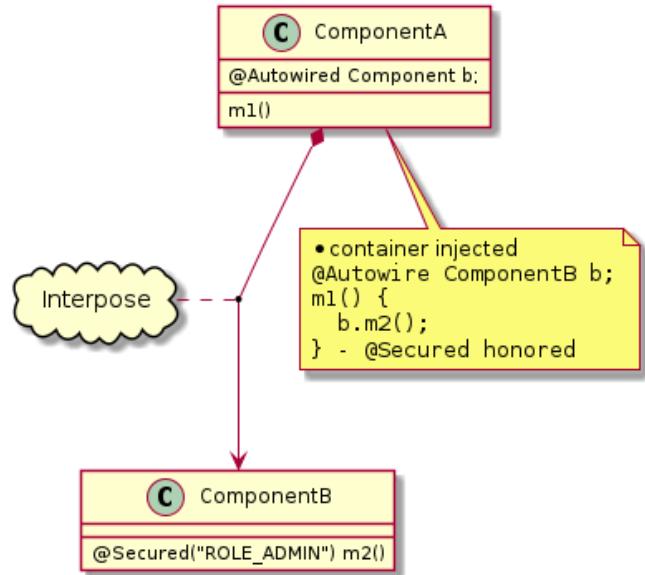


Figure 149. Container Injected Method Boundary

364.2. 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.

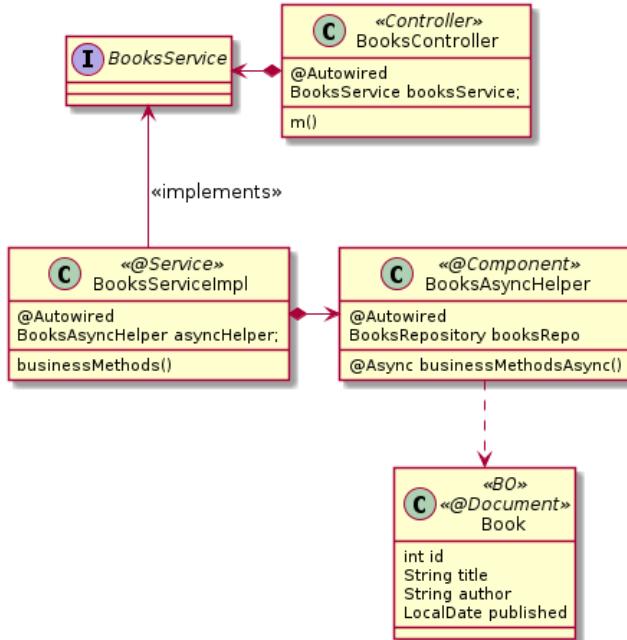


Figure 150. Single Service Expressed as Multiple Components

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`.

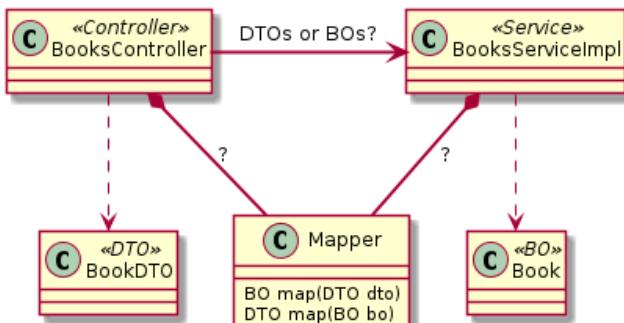
Conceptual Services may be broken into Multiple Physical Components

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 365. 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 151. BO/DTO Interface Decisions

365.1. 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 `@Controller` 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.

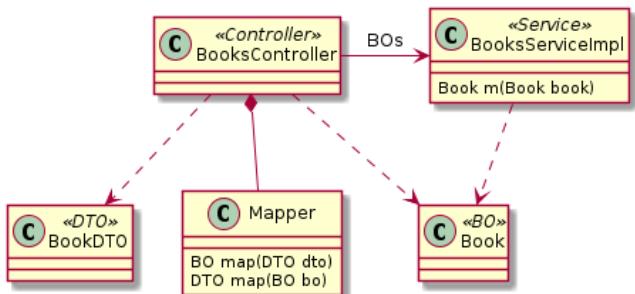
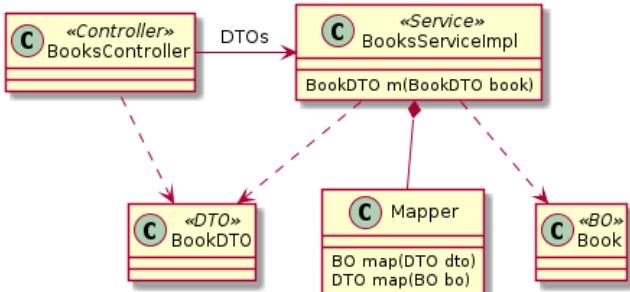


Figure 152. API Maps DTO to BO for Service Interface

365.2. @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 `@Controller` and all `@Service` clients work with DTOs.

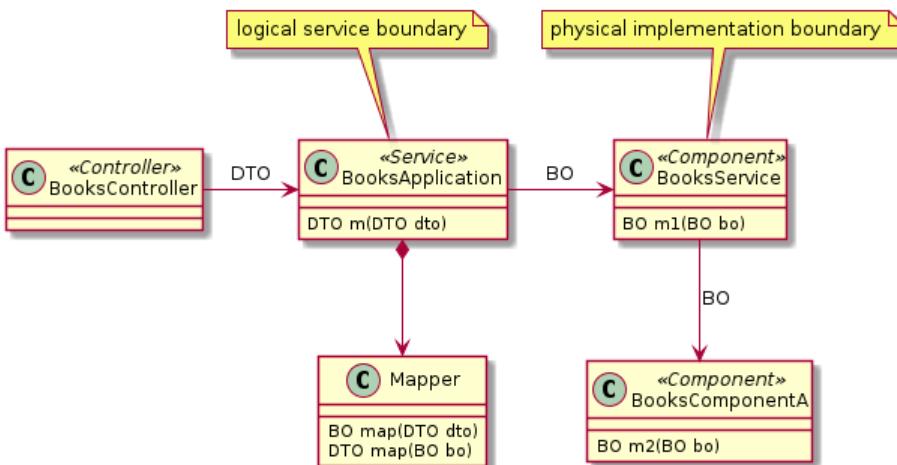


- Benefit: `@Service` fully encapsulates implementation and exchanges information using DTOs designed for interfaces.
- Drawback: BOs go through a translation when passing from `@Service` to `@Service` directly.

Figure 153. Service Maps DTO in Service Interface to BO

365.3. 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 366. Implementation Details

With architectural decisions understood, lets take a look at some of the key details of the end-to-end application.

366.1. 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  
@AllArgsConstructor  
public class Book {  
    @Id  
    private String id;  
    @Setter  
    @Field(name="title")  
    private String title;  
    @Setter  
    private String author;  
    @Setter  
    private LocalDate published;  
}
```

366.2. 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.

```
package info.ejava.examples.db.repo.jpa.books.dto;  
...  
@JacksonXmlRootElement(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

366.2.1. 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)
private LocalDate published;

public static class LocalDateJaxbAdapter extends XmlAdapter<String, LocalDate> {
    @Override
    public LocalDate unmarshal(String text) {
        return LocalDate.parse(text, DateTimeFormatter.ISO_LOCAL_DATE);
    }
    @Override
    public String marshal(LocalDate timestamp) {
        return DateTimeFormatter.ISO_LOCAL_DATE.format(timestamp);
    }
}
```

366.3. 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"
}
```

366.4. 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 xmlns="">To a God Unknown</title>
    <author xmlns="">Rudolf Harris</author>
    <published xmlns="">2019-11-22</published>
</book>
```

```
<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>
```

366.5. 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. The DTO classes have a direct dependency on their non-DTO siblings.
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.
3. Although technically I could use either, 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.

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.

366.5.1. 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.

366.5.2. 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

366.5.3. 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`

366.5.4. 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"/>
```

366.6. 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.

366.6.1. 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

366.6.2. 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()));
    }
}

```

366.6.3. 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 Spring Data `Page<T>` to DTO

366.6.4. 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

```
BooksPageDTO pageDTO = request.exchange()  
    .expectStatus().isOk()  
    .returnResult(BooksPageDTO.class)  
    .getResponseBody().blockFirst();  
  
Page<BookDTO> page = pageDTO.toPage(); ①  
Pageable pageable = ... 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 367. 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.

367.1. 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;
    }
    ...
}
```

367.2. 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 368. Service Tier

The BooksService `@Service` encapsulates the implementation of our management of Books.

368.1. 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

368.2. 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.

368.3. `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

368.4. `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 369. 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.

369.1. 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

369.2. 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

369.3. 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 370. 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 © 2021 jim stafford (jim.stafford@jhu.edu)

Chapter 371. Introduction

This lecture contains several "how to" aspects of building and deploying a Docker image to Heroku with Postgres or Mongo database dependencies.

371.1. 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

371.2. 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 372. 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.

372.1. 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

372.2. 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 373. 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
```

373.1. 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.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}
```

373.2. 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.

373.3. 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

373.4. 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
adoptopenjdk:14-jre-hotspot \
/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

373.5. 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
```

373.6. 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 374. 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.

374.1. 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 eliminate the glob from matching other JAR files like `.sources.jar`. We can also use `.dockerignore` file. 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 adoptopenjdk:14-jre-hotspot as builder
WORKDIR application
ARG JAR_FILE=target/*SNAPSHOT.jar ①
COPY ${JAR_FILE} application.jar
RUN java -Djarmode=layer tools -jar application.jar extract

FROM adoptopenjdk:14-jre-hotspot
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.JarLauncher"]
```

① need to make sure we do not match more than a single JAR file if search performed

② added a filter script to break certain environment variables into separate properties

374.2. 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 both snapshot and release-based 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}.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:6.0.0-SNAPSHOT
[INFO]
[INFO] Detected build of image with id dfe2383f7f68
...
[INFO] Successfully built dockercompose-votes-svc:6.0.0-SNAPSHOT
[INFO] -----
[INFO] BUILD SUCCESS
```

Chapter 375. Heroku Deployment

The following are the basic steps taken to deploy the Docker image to Heroku.

375.1. 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
```

375.2. 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
```

375.3. 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 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
```

375.4. 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
```

375.5. 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
\\ / _ ' _ _ _ ( _ ) _ _ _ \ \ \
( ( ) \ _ _ | ' _ | ' _ | ' _ \ \ _ ' | \ \ \
\ \ / _ _ ) | ( _ ) | | | | | | ( _ | | ) ) ) )
' | _ _ | . _ | _ | _ | _ \ _ , | / / / /
=====|_|=====|_/_=/_/_/_/
:: Spring Boot ::      (2.4.2)
The following profiles are active: production ①
```

① make sure the application is running the correct profile

Chapter 376. 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

Assignment 5: DB

copyright © 2021 jim stafford (jim.stafford@jhu.edu)

This assignment is broken up into three mandatory sections and an optional BONUS section for those that need extra credit.

The first two mandatory sections functionally work with Spring Data Repositories outside the scope of the Racer Registration workflow. You will create a "service" class that is a peer to your Racer Registration 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.

In the forth, optional BONUS section—you may switch technology sections and implement Races or Racers using a Spring Data Repository. However, in this case—it would be done using only the existing legacy interfaces.

Chapter 377. Assignment 5a: Spring Data JPA

377.1. Database Schema

377.1.1. 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

377.1.2. Overview

In this portion of the assignment you will be defining, instantiating, and performing minor population of a database schema for Racer Registration. We will use a single, flat database design.

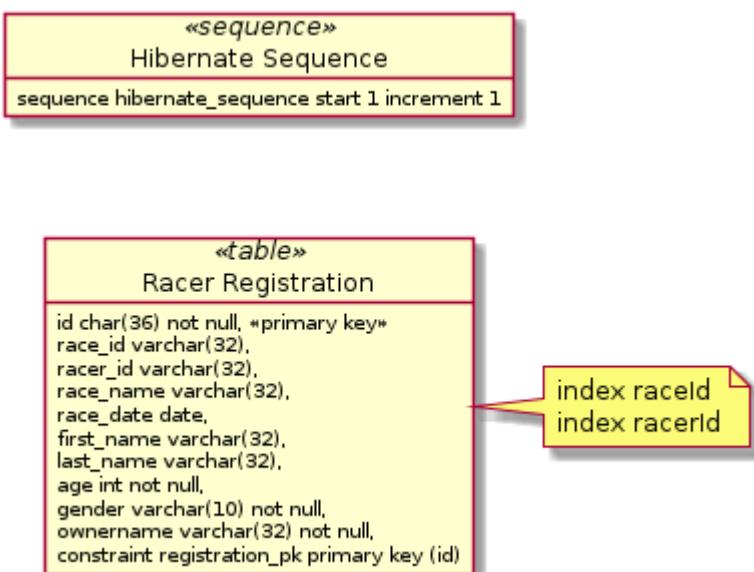


Figure 154. Racer Registration Schema

I have shown the creation of a sequence despite choosing to use a `char(36)` primary key for the table. Please keep the sequence in your schema as sequences are commonly needed in RDBMS solutions.

Use char(36) to allow consistency with Mongo portion of assignment



I have chosen for you to use the char-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 ≤ 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/env 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.



```
@SpringBootTest(classes={DbRaceApp.class})
@ActiveProfiles(profiles={"test"}, resolver = TestProfileResolver.class)
//@ActiveProfiles(profiles={"test", "postgres"})
public class Jpa5a_SchemaTest {
```

377.1.3. Requirements

1. Add default profiles in `application.properties` in the `src/main` tree. This will activate the security settings you finished up with on the previous assignment.

application.properties

```
spring.profiles.active=authorities,authorization
```

2. Configure Database Properties

- a. set the default database to `h2` and activate the console

application.properties

```
#default test database
spring.datasource.url=jdbc:h2:mem:race
spring.h2.console.enabled=true
```

- b. set logging level for JPA/SQL-related DEBUG

application.properties

```
spring.jpa.show-sql=true
logging.level.org.hibernate.type=trace
```

- c. add a `postgres` profile in `src/main` tree to optionally connect to an external Postgres server versus the in-memory H2 server

application-postgres.properties

```
spring.datasource.url=jdbc:postgresql://localhost:5432/postgres
spring.datasource.username=postgres
spring.datasource.password=secret
```

3. Create a set of SQL migrations below `src/main/resources/db/migration` that will define the database schema

- a. create a SQL migration file to define the base Racer Registration schema. This can be hand-generated or metadata-generated once the `@Entity` class is later defined
 - i. define a sequence called `hibernate_sequence`
 - ii. define a Racer Registration table
 - A. the tests assume the table name will be `race_registration`.
 - B. use the `id` field as a primary key. Make this a char-based column type of at least 36 characters (`char(36)`) to be able to host a UUID string
 - C. define column constraints for size and non-null
 - iii. account for when the table(s)/sequence(s) already exist by defining a DROP before creating
- b. Create a SQL migration file to add indexes
 - i. define a non-unique index on `raceId`
 - ii. define a non-unique index on `racerId`
- c. Create a SQL migration file to add one row in the Racer Registration table



Refer to the [JPA songs example](#) for a schema example



`CURRENT_DATE` can be used to generate a value for `race_date`

You can manually test schema files by launching the Postgres client and reading the SQL file in from stdin



```
docker-compose exec -T postgres psql -U postgres < (path to file)
```

- d. Place vendor-neutral SQL in `db/migration/common` and vendor-specific in `db/migration/{vendor}` below `src/main/resources/` — where `{vendor}` is either `h2` or `postgres`.

```
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.

- e. Add a Flyway migration file search path property to identify the location of the migration files:

```
spring.flyway.locations=classpath:db/migration/common,classpath:db/migration/{vendor}
```

4. Configure the application to establish a connection to the database and establish a DataSource
- declare a dependency on `spring-boot-starter-data-jpa`
 - declare a dependency on the `h2` database driver for default testing
 - declare a dependency on the `postgresql` database driver for optional production-ready testing
 - declare the database driver dependencies as `scope=runtime`



See `jpa-song-example pom.xml` for more details on declaring these dependencies.

5. Configure Flyway so that it automatically populates the database schema
- declare a dependency on the `flyway-core` schema migration library
 - declare the Flyway dependency as `scope=runtime`



See `jpa-song-example pom.xml` for more details on declaring this plugin

6. Define a JUnit unit integration test to (provided / enable)
 - a. verify the `DataSource` can be injected
 - b. verify the table exists
7. Package the JUnit test case such that it executes with Maven as a surefire test (provided)

377.1.4. 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

377.1.5. Additional Details

377.2. Entity/BO Class

377.2.1. 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

377.2.2. Overview

In this portion of the assignment you will be creating an @Entity/Business Object for a Racer Registration, mapping that to a table, and performing CRUD actions with an EntityManager.

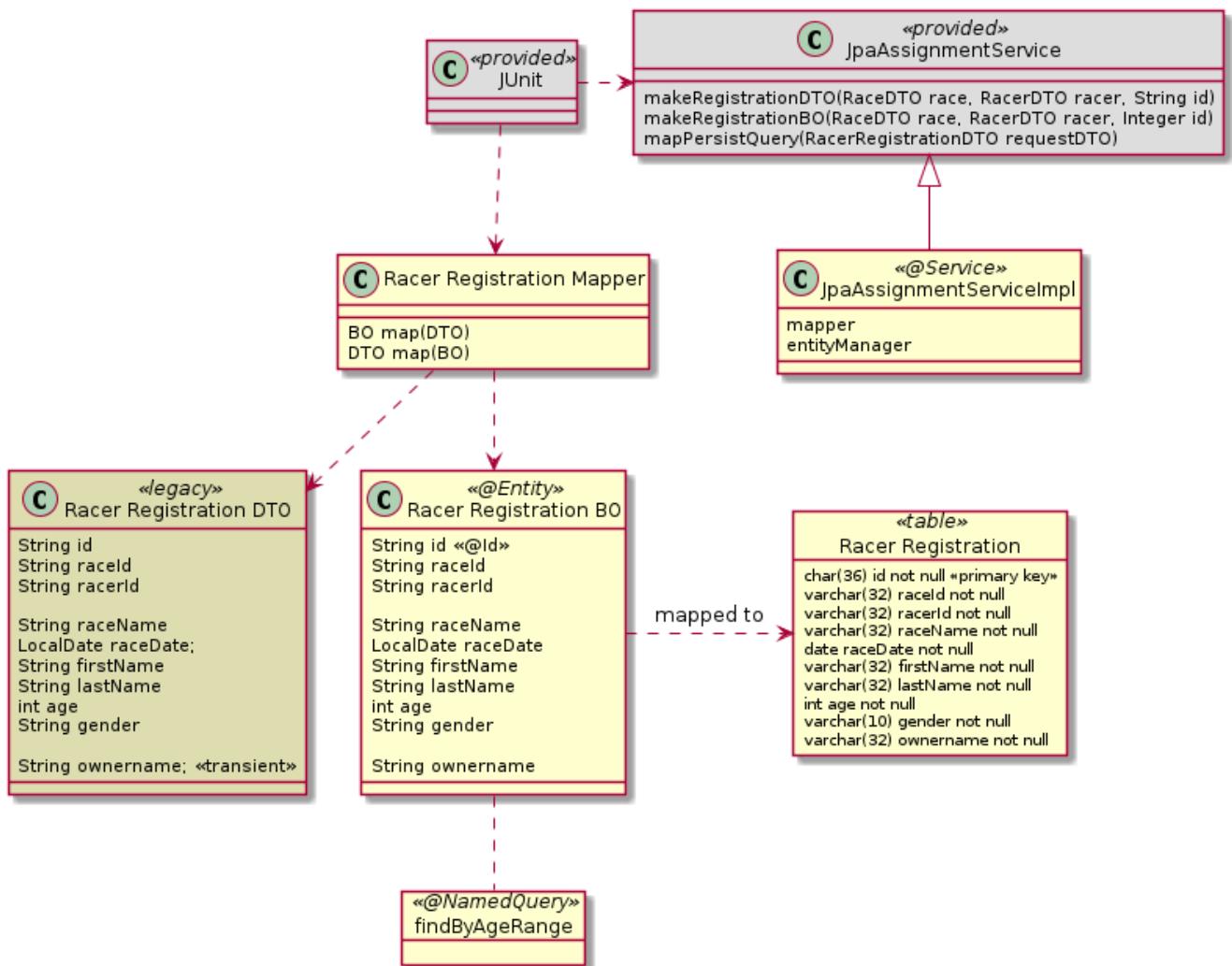


Figure 155. Racer Registration Entity

A **JpaAssignmentService** interface has been defined with helper methods that you will implement in the **JpaAssignmentServiceImpl** class. These helper methods

1. help isolate the project-specific implementation details from the JUnit test
2. give you a specific place to implement required logic and persistence calls

377.2.3. Requirements

1. Create an Business Object (BO) class that represents the Racer Registration and will be mapped to the database. A **RacerRegistrationBO** class has been provided for this. You must complete the details.
 - a. identify the class as a JPA **@Entity**
 - b. identify a String primary key field with JPA **@Id**
 - c. supply a default constructor

- d. supply other constructs as desired to help use and interact with this business object
- e. 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`

```
@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 Racer Registration BO and DTO. A `RacerRegistrationMapper` class has been provided for this. You must complete the details.
 - a. map from BO to DTO
 - b. map from DTO to BO
3. Fill in the mapper details of the `JpaAssignmentServiceImpl` class
 - a. supply a factory method that will create an instance of a Racer Registration BO from a `RaceDTO`, `RacerDTO`, and `id` value. The registration `ownername` should be set to the `username` of the `RacerDTO`.
 - b. supply a factory method that will create an instance of a Racer Registration DTO from a `RaceDTO`, `RacerDTO`, and `id` value. The registration `ownername` should be set to the `username` of the `RacerDTO`.



These are just helper methods to better isolate your details of DTO construction from the JUnit tests.

4. Created a `@NamedQuery` in the Racer Registration BO class that will find all registrations between a min and max age (inclusive)
 - a. `min` and `max` are variable integer values passed in at runtime
 - b. order the results by `id` ascending
 - c. name the query "RacerRegistrationBO.findByAgeRange"

You may use the following query to start with



```
select r from RacerRegistrationBO r where r.age between :min and :max
```

5. Supply the details for a `mapPersistQuery` method in the `JpaAssignmentServiceImpl` class that must perform the following
 - i. accept a Racer Registration DTO
 - ii. map the DTO to a Racer Registration BO (using your mapper)
 - iii. persist the BO
 - iv. query the database using the `@NamedQuery` supplied above to locate matching rows that will include the persisted BO
 - v. map the BO list returned from the query to a list of DTOs (using your mapper)
 - vi. return the list of DTOs
 - vii. all actions of the method must be performed under the scope of a single, active transaction
6. *Create a JUnit unit integration test (provided / activate)*
 - a. *inject an EntityManager and mapper to use for test methods (provided)*
 - b. *verify the ability to map from DTO to BO (provided)*
 - c. *verify the ability to map from BO to DTO (provided)*
 - d. *verify the ability to persist a BO and locate using a query (provided)*
7. *Package the JUnit test case such that it executes with Maven as a surefire test (provided)*

377.2.4. Grading

Your solution will be evaluated on:

1. inject an EntityManager to perform actions on a Persistence Unit and database
 - a. whether a EntityManager was successfully injected into the JUnit test
2. map a simple `@Entity` class to the database using JPA mapping annotations
 - a. whether a new Racer Registration class was created for mapping to the database
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 Racer Registration 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

377.2.5. Additional Details

377.3. JPA Repository

377.3.1. 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

377.3.2. Overview

In this portion of the assignment you will define a JPA Repository to perform basic CRUD and query actions.

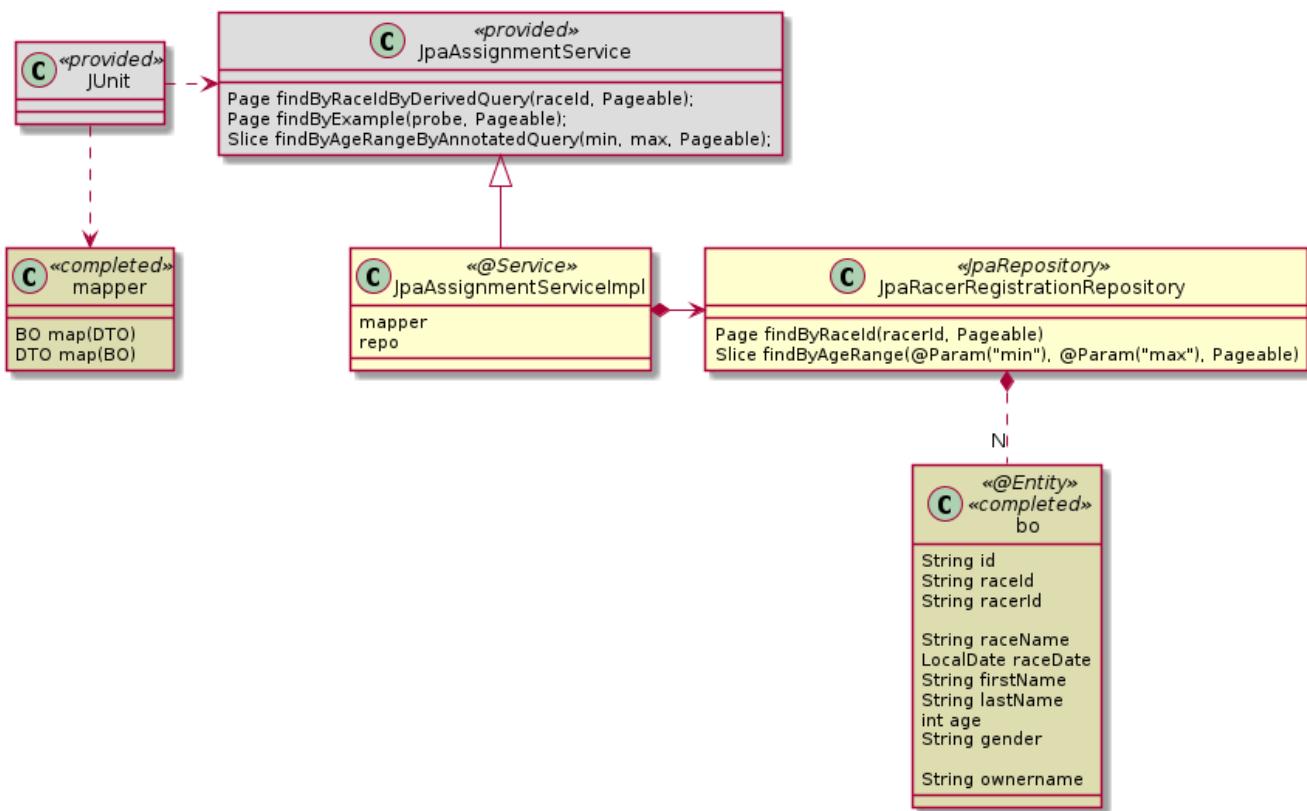


Figure 156. Racer Registration Entity

A `JpaAssignmentService` interface has been defined with some helper methods for the JUnit tests. Implement the details within the `JpaAssignmentServiceImpl` and `JpaRacerRegistrationRepository` to complete some of the requested use cases.

377.3.3. Requirements

1. define a Racer Registration JPARepository that can support basic CRUD and complete the queries defined below. A `JpaRacerRegistrationRepository` interface has been provided to perform this function.
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 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 `@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 the `JpaAssignmentServiceImpl`. This will be enough to tell you whether the Repository is properly defined and registered with the Spring context.
4. implement the `findByIdRaceIdByDerivedQuery` method details which must
 - a. accept a raceId 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 Racer Registration 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. Example: `int age`.

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 "RacerRegistrationBO.findByAgeRange" `@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
```

7. implement the JUnit test so that it executes during the Maven surefire phase (provided / activate)

377.3.4. 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

377.3.5. Additional Details

Chapter 378. Assignment 5b: Spring Data Mongo

378.1. Mongo Client Connection

378.1.1. 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 an MongoOperations/MongoTemplate instance to perform actions on a database

378.1.2. 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.

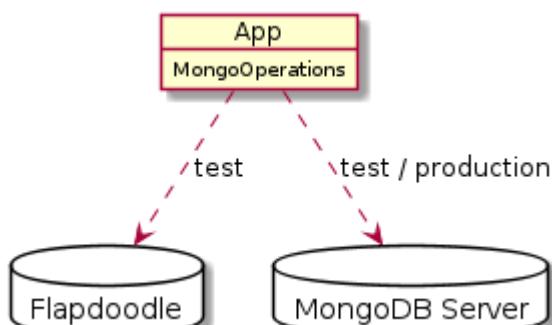


Figure 157. Mongo Client Connection

Postgres access with Docker/Docker Compose

If you have Docker/Docker Compose, you can instantiate a MongoDB instance using the scripts in the ejava-springboot/env 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(classes={DbRaceApp.class})
@ActiveProfiles(profiles = "test", resolver = TestProfileResolver.
class)
//@ActiveProfiles(profiles = {"test", "mongodb"}, resolver =
TestProfileResolver.class)
public class Mongo5a_ClientTest {
```

378.1.3. Requirements

1. Configure database properties.
 - a. set logging level for MongoDB-related DEBUG

application.properties

```
logging.level.org.springframework.data.mongodb=DEBUG
```

- b. add a `mongodb` profile in `src/main` tree to optionally connect to an external MongoDB server versus the internal Flapdoodle test server

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=a
dmin
```



Configure via Individual Properties or Compound URL

Spring Data Mongo has the capability to set configuration properties a value at a time or via one, compound URL.



Flapdoodle will be Default Database during Testing

Flapdoodle will be the default during testing unless explicitly deactivated

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.

3. Define a JUnit unit integration test to (provided / enable)
 - a. verify the `MongoOperations/MongoTemplate` can be injected
 - b. verify the communication with the database
4. Package the JUnit test case such that it executes with Maven as a surefire test (provided)

378.1.4. 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

378.1.5. Additional Details

378.2. Mongo Document

378.2.1. 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

378.2.2. Overview

In this portion of the assignment you will be creating a `@Document`/Business Object for a Racer Registration, mapping that to a collection, and performing CRUD actions with a `MongoOperations/MongoTemplate`.

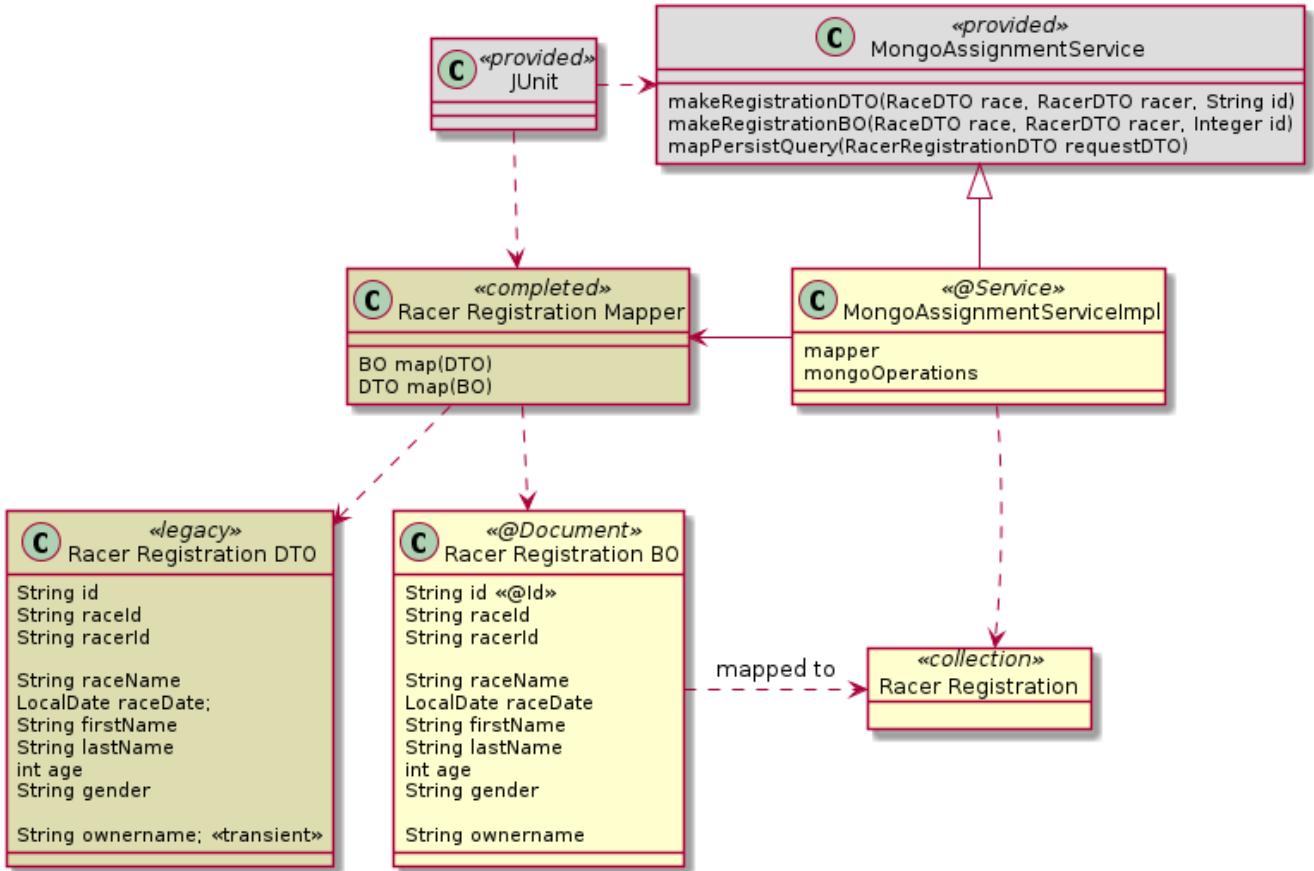


Figure 158. Mongo Document

Reuse BO and Mapper classes



It has been designed and expected that you will be able to re-use the same Racer Registration BO and Mapper classes. 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.

378.2.3. Requirements

1. Create an Business Object (BO) class that represents the Racer Registration and will be mapped to the database. A `RacerRegistrationBO` class has been provided for this. You must complete the details.
 - a. identify the class as a Spring Data Mongo `@Document`
 - b. identify a String primary key field with Spring Data Mongo `@Id`

This is a different `@Id` annotation than the JPA `@Id` annotation.

- c. supply a default constructor
2. Fill in the mapper details of the `MongoAssignmentServiceImpl` class
 - a. supply a factory method that will create an instance of a Racer Registration BO from a `RaceDTO`, `RacerDTO`, and `id` value. The registration `ownername` should be set to the `username` of the `RacerDTO`.
 - b. supply a factory method that will create an instance of a Racer Registration DTO from a

`RaceDTO`, `RacerDTO`, and `id` value. The registration `ownername` should be set to the `username` of the `RacerDTO`.



These are just helper methods to better isolate your details of DTO construction from the JUnit tests. You likely can reuse the same code implemented in the JPA portion of the assignment.

3. Supply the details for a `mapPersistQuery` method in the `MongoAssignmentServiceImpl` class that must perform the following

- i. accept a Racer Registration DTO
- ii. map the DTO to a Racer Registration BO (using your mapper)
- iii. persist the BO
- iv. query the database to locate matching rows that will include the persisted BO



You may use the injected `MongoOperations` client find command, a query, and the `RacerRegistrationBO.class` as a request parameter

You may make use of the following query



```
Query.query(Criteria  
    .where("age")  
    .gte(bo.getAge()-1)  
    .lte(bo.getAge()+1))
```

- v. map the BO list returned from the query to a list of DTOs (using your mapper)
 - vi. return the list of DTOs
4. *Create a JUnit unit integration test (provided / activate)*
- a. *inject a MongoOperations and mapper to use for test methods (provided)*
 - b. *verify the ability to persist a BO and locate using a query (provided)*
5. *Package the JUnit test case such that it executes with Maven as a surefire test (provided)*

378.2.4. 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`

378.2.5. Additional Details

378.3. Mongo Repository

378.3.1. 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

378.3.2. Overview

In this portion of the assignment you will define a Mongo Repository to perform basic CRUD and query actions.

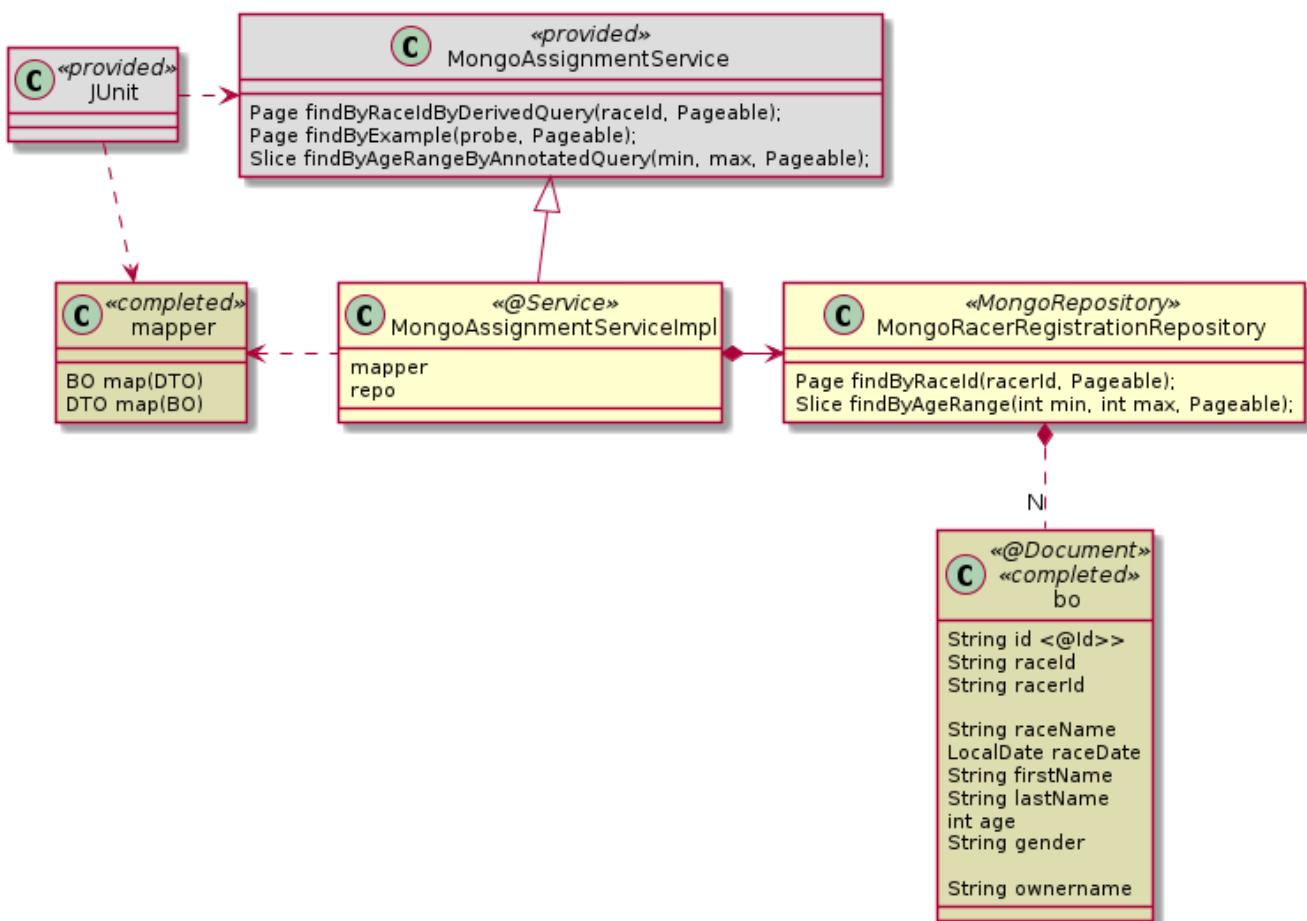


Figure 159. Mongo Repository

378.3.3. Requirements

1. define a Racer Registration MongoRepository that can support basic CRUD and complete the queries defined below. A `MongoRacerRegistrationRepository` interface has been provided to perform this function.
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 `MongoAssignmentServiceImpl`. This will be enough to tell you whether the Repository is properly defined and registered with the Spring context.
4. implement the `findByIdRaceIdByDerivedQuery` method details which must
 - a. accept a raceId 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 Racer Registration 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. Example: `int age`.

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 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="{ age : {$gte:?0, $lte: ?1} }" ①
```

① min is position 0 and max 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}"
```

7. implement the JUnit test so that it executes during the Maven surefire phase (provided / activate)

378.3.4. 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 ..whether queries were performed with sorting and paging

378.3.5. Additional Details

Chapter 379. Assignment 5c: Spring Data Application

379.1. API/Service/DB End-to-End

379.1.1. 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

379.1.2. Overview

In this portion of the assignment you will be taking the elements of an application that you have worked on either together or independently and integrate them into an end-to-end application from the API, thru services and security, thru the repository, to the database and back.

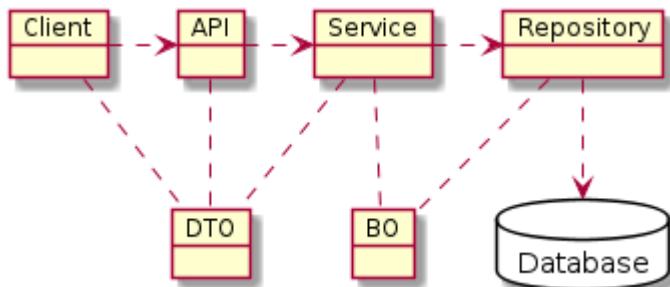


Figure 160. API/Service/DB End-to-End

The fundamental scope of the assignment is to perform existing Racer Registration 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 Racer Registration Service and Repository implementation classes with implementations that are based upon your selected repository technology
- update the controller and service interfaces to address paging
- leave the Races and Racers implementation as in-memory repositories

379.1.3. 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 Racer Registration 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 queries must accept Pageable and return Page
- c. the service should
 - i. accept DTO types as input and map to BOs for interaction with the Repository
 - ii. map BO types to DTOs as output



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. Update/add controller query methods to

- a. accept pageNumber, pageSize, and sort properties
- b. return a DTO representation of a Page<DTO> to include page specs



There is a [DTO Paging framework](#) within common that is used by the [JPA songs](#) and [Mongo books](#) examples. It includes XML constructs that you may need to provide, but only JSON will be required in your service implementation.

- c. Add additional query method that selects Race Registrations for a specific raceId
 - i. add a representation of pageable parameters and the raceId to the input controller input
 - ii. return a page representation of the DTOs that meet the criteria and an expression of the paging criteria



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.

4. Add a JUnit unit integration test that
 - a. populates the database with rows/documents
 - b. demonstrates the query and paging
5. Turn in a source tree with complete Maven modules that will build web application.

379.1.4. Grading

Your solution will be evaluated on:

1. implement a service tier that completes useful actions
 - a. whether you successfully implemented a query for Racer Registrations for a specific raceId
 - 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

379.1.5. Additional Details

Chapter 380. Assignment 5d: Bonus

If you feel you got a slow start to the semester and are now finally getting on a role, the following is an optional BONUS assignment for your consideration. If you complete this bonus assignment successfully and need a ~half-grade boost to get you into the next tier you may want to read on.

380.1. Races/Racers Alternate Repository

380.1.1. Purpose

In this portion of the optional BONUS assignment, you will demonstrate your knowledge of integrating a Spring Data Repository into an end-to-end application, accessed through an API of the opposite technology than you chose for Racer Registrations.

You will also demonstrate the ability to weave a new component implementation into the Spring context to replace an existing in-memory Repository.

380.1.2. Races/Racers Alternate Repository

In this portion of the optional BONUS assignment you will be replacing the in-memory Races or Racers Repository with a version that uses the database technology you did not choose for the Application portion of this overall assignment.

You do not need to change any interfaces, which means your expression of Pageable and Page might be quite limited or non-existent.

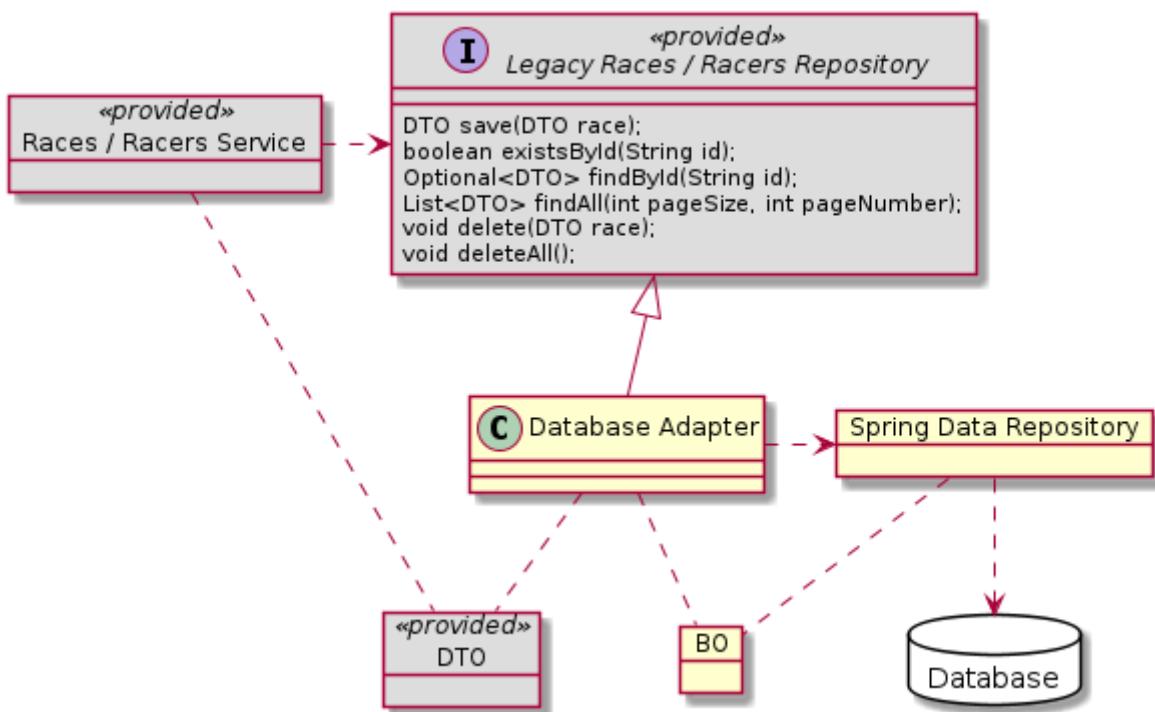


Figure 161. API/Service/DB End-to-End

The fundamental scope of the assignment is to replace the existing Races or Racers Repository Impl classes with classes that adapt the legacy calls to the Spring Data Repository. There are six nearly

calls that have to be replaced.

If you implemented Racer Registration End-to-End with MongoRepository, you will implement the adapter in this assignment using JpaRepository. If you implemented Racer Registration End-to-End with JpaRepository, you will implement the adapter in this assignment using MongoRepository.

Unlike the Racer Registration End-to-End assignment, there are no interface changes requested or desired here. You are replacing a component implementation without breakage.

380.1.3. Requirements

1. Select a database implementation choice (JpaRepository or MongoRepository) that is the opposite of what was chosen for Racer Registration.



You should have all dependencies already in-place.

2. Add Races or Racers Repository Adapter that will
 - a. accept DTOs as a part of the legacy interface
 - b. map the DTO to a BO class
 - c. interact with a new Spring Data Repository component that you will create
 - d. map any BO results to DTOs
3. Add Races or Racers Spring Data Repository interface definitions that will satisfy the requirements of the Races or Racers Adapter
4. Add a `@Bean` factory that will take priority over the legacy Races or Racers `@Bean` factory supplied by auto-configuration.
5. Provided a JUnit unit integration test that will demonstrate a successful registration — from Race/Racer creation to Racer Registration completion, using your new Adapter.
6. Turn in a source tree with complete Maven modules that will build web application.

380.1.4. Grading

Your solution will be evaluated on:

1. whether you were able to surgically inject the Adapter as a new component implementation for the end-to-end Races or Racers calls.
2. whether you were able to implement persistence using a Spring Data Repository that was opposite of what you used for the Racer Registration End-to-End.

380.1.5. Additional Details

Bean Validation

copyright © 2021 jim stafford (jim.stafford@jhu.edu)

Chapter 381. 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 (personDTO==null) {  
        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 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

381.1. 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

381.2. 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 382. Background

[Bean Validation](#) is a standard that originally came out of Java EE/SE as JSR-330 (1.0) in the 2009 timeframe and later updated with JSR-349 (1.1) in 2013 and [JSR-380 \(2.0\)](#) in 2017. 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. ^[76]

Implementations include:

- [Hibernate Validator](#)
- [Apache BVal](#)

Hibernate Validator was the original and current reference implementation and used within Spring Boot today.

^[76] ["Bean Validation specification"](#), Gunnar Morling, 2017

Chapter 383. 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:2.4.2:compile
[INFO] |  +- org.glassfish:jakarta.el:jar:3.0.3:compile ③
[INFO] |  \- org.hibernate.validator:hibernate-validator:jar:6.1.7.Final:compile ②
[INFO] |    +- jakarta.validation:jakarta.validation-api:jar:2.0.2:compile ①
[INFO] |    +- org.jboss.logging:jboss-logging:jar:3.4.1.Final:compile
[INFO] |    \- com.fasterxml:classmate:jar:1.5.1:compile
```

① overall Bean Validation API

② Bean Validation API reference implementation from Hibernate

③ regular expression implementation for regular expression constraints

Chapter 384. Declarative Constraints

At the core of the Bean Validation API are declarative constraint annotations we can place directly into the source code.

384.1. 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 javax.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.

384.2. 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 25. 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.

384.3. 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 javax.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 aParameter.toString();
    }
}
```

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 385. Programmatic Validation

To work with the Bean Validation API directly, our initial goal is to obtain a standard `javax.validation.Validator` instance.

Programmatic Validation Requires Validator

```
import javax.validation.Validator;  
...  
Validator validator;
```

This can be obtained manually or through injection.

385.1. Manual Validator Instantiation

We can obtain 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. We will come back to the `configure()` method later. 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.

Standard javax.validation.Validator Interface

```
import javax.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

385.2. 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;
```

385.3. Customizing Injected Instance

If we want the best of both worlds and have some customizations to make, we can define a @Bean to 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

385.4. 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;
    ...
}
```

385.5. 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
for (ConstraintViolation v: violations) {
    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 where 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
```

Property and Value Validation



We can also validate a specific object property (`validateProperty()`) or a value against the definition of a property (e.g., `validateValue()`).

385.6. 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 aParameter.toString();  
    }  
}
```

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.

Obtain 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.

385.7. 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);
```

385.8. 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());
```

385.9. 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 386. 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 aParameter.toString();  
    }  
}
```

Validation result using Argument Position

```
then(violation.getPropertyPath().toString()).isEqualTo("aMethod.arg0.aValue");①
```

① By default, argument position supplied (`arg0`) — not argument name

There are two ways to solve this

386.1. Add -parameters to Java Compiler Command

The first way to solve this would be to add the `-parameter` 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>
    <compilerArgs>-parameters</compilerArgs>
  </configuration>
</plugin>
```

386.2. 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

386.3. ParameterNameProvider

The following snippets shows 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.

```
import javax.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

386.4. 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 `javax.inject.Named` to supply a textual name of my choice.

```
import javax.inject.Named;
...
private static class AService {
    @NotNull
    public String aMethod(@NotNull @Valid @Named("aParameter") AClass aParameter) {①
        return aParameter.toString();
    }
}
```

① `@Named` annotation is providing a name to use for `MyParameterNameProvider`

386.5. 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()) ?
    argName(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 387. Graphs

Constraint validation has the ability to 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 one of the child references is annotated with `@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

387.1. 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

387.2. Graph Traversal

Adding the `@Valid` annotation to an object reference activates traversal to and validation of the child instance. This is continuous to grandchildren, etc.

The following snippet shows an example of a parent who's validation traverses to the child because of the `@Valid` annotation.

Validation Traversal to Child

```
import javax.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 388. 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 `javax.validation.groups.Default` group and were evaluated by default because no groups were specified in the call to `validate()`.

We can define our own custom groups using interfaces.

388.1. 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 javax.validation.groups.Default;
...
public interface PocValidationGroups { ③
    public interface Create{}; ①
    public interface CreatePlusDefault extends Create, Default{}; ②
    public 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 not required, Used in example to make purpose and source of group obvious

388.2. Applying Groups

We can assign specific groups to constraints individually.

In the following example,

- `@NotNull 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`

388.3. 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.

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`.

388.4. 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 is was assigned to was part of the validation command.

Validation Group Applied Example

```
//given
...
String expectedError="id:cannot be specified for create";
//when
violations = validator.validate(invalidForCreate, CreatePlusDefault.class); ①
//then
then(violations).hasSize(1); ②
then(errors(violations)).contains(expectedError); 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 389. 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.

389.1. 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 simple to validate
- `@Email email` is thought to be a more detailed validation
- the remaining validations have not been addressed by this classification

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

```
public interface Create{};  
public interface SimplePlusDefault extends Default {}  
public interface DetailedOnly {}
```

389.2. 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

389.3. 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

389.4. 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 390. 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 a 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 `@Validation` annotation is new. The `@Validation` annotation triggers Spring to apply Bean Validation to calls made on the type (interface or class).

Component Interface

```
import org.springframework.validation.annotation.Validated;  
  
import javax.inject.Named;  
import javax.validation.constraints.Min;  
  
② @Validated  
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

390.1. 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

```
① @Component  
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.

```
@Autowired  
private ValidatedComponent component;
```

390.2. 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 and throws a `javax.validation.ConstraintViolationException` before making the call. The `Set<ConstraintViolation>` can be obtained from the exception. At that point we have returned to some familiar territory we covered with programmatic validation.

Injected Component Validation by Spring

```
import javax.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

390.3. 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

390.4. 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 added any additional constraint beyond what the interface defined—we will get a `ConstraintDeclarationException` thrown. The Bean Validation Specification [describes it](#) as following the [Liskov Substitution Principle](#)—where anything that is a sub-type of `T` can be inserted in place of `T`. Said more specific to Bean Validation—a sub-type or implementation class method cannot add more restrictive constraints to call.

```
@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)

390.5. 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.

```
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

390.6. 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
    public 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

390.7. 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 391. 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;
```

391.1. 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 {};  
    int age() default 0;  
    int tzOffsetHours() default 0;  
  
    @Documented  
    @Retention(RUNTIME)  
    @Target({ METHOD, FIELD, ANNOTATION_TYPE, PARAMETER, TYPE_PARAMETER })  
    public @interface List {  
        MinAge[] value();  
    }  
}
```

391.2. @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 162. 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 163. `@PersonHasName` Annotation Javadoc

391.3. `@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 26. Annotation `@Target ElementType`

<i>ElementType.FIELD</i>	<i>ElementType.METHOD</i>
<pre>@MinAge LocalDate dob;</pre>	<pre>@MinAge LocalDate getDob(); @MinAge void add(LocalDate dob, LocalDate dateOfHire);①</pre> <p>① <code>@MinAge</code> being used as cross-param constraint here</p>
define validation on a Java attribute within a class	define validation on a return type or cross-parameters of a method
<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 dobs; }</pre>	<pre>class Person { LocalDate dobs; @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>	

391.4. @Retention

`@Retention` is used to determine the lifetime of the annotation.

Annotation @Retention

```
@Retention(  
    //SOURCE - annotation discarded by compiler  
    //CLASS - annotation available in class file but not loaded at runtime - default  
    RetentionPolicy.RUNTIME //annotation available thru reflection at runtime  
)
```

Bean Validation should always use **RUNTIME**

391.5. @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 })  
    public @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() {
```

391.6. @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**

391.6.1. 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 {};  
    ...
```

391.7. @MinAge-specific Properties

Each individual 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;  
    ...
```

391.8. 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 optional provide additional or custom details using the context.

391.9. 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) { ... }
```

391.10. 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());  
    }  
}
```

391.11. 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`.

```
@Override  
public boolean isValid(LocalDate dob, ConstraintValidatorContext context) {  
    if (dob!=null) { //assume null is valid and use @NotNull if it should not be  
        final LocalDate now = LocalDate.now(zoneOffset);  
        final int currentAge = Period.between(dob, now).getYears();  
        return currentAge >= minAge;  
    }  
    return true;  
}
```



Treat Null Values as Valid

Null values should be considered valid and independently constrained by `@NotNull`.

391.12. 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.getDefaultConstraintMessageTempl  
ate())  
    .addParameterNode(1)  
    .addConstraintViolation()  
    .buildConstraintViolationWithTemplate(context  
    .getDefaultConstraintMessageTemplate())  
    .addParameterNode(2)  
    .addConstraintViolation();  
//the following removes default-generated message  
//context.disableDefaultConstraintViolation(); ①
```

① make this call to eliminate 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 392. 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

392.1. 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 apparent.

Cross-Parameter Annotation

```
@Documented  
@Constraint(validatedBy = ConsistentNameParameters.ConsistentNameParametersValidator  
.class )  
@Target({ElementType.METHOD, ElementType.CONSTRUCTOR})  
@Retention(RetentionPolicy.RUNTIME)  
public @interface ConsistentNameParameters {
```

392.2. @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.

392.3. 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

392.4. 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 393. Web API Integration

393.1. 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.

393.2. 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 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:
javax.validation.ConstraintViolationException:
createPOC.person.id: cannot be specified for create",
  "date" : "2021-07-01T14:58:48.777269Z" }
```

① INTERNAL_SERVER_ERROR status is mis-leading—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 on the client needs to be corrected.

393.3. 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 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.

```
@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

393.4. 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",
  "date" : "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.

393.5. 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.

```
@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

393.6. 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.

393.7. 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`.

```
@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

        List<String> fieldMsgs = ex.getFieldErrors().stream() ②
            .map(e -> e.getObjectName() + ":" + e.getField() + ": " + e.getDefaultMessage())
            .collect(Collectors.toList());
        List<String> globalMsgs = ex.getGlobalErrors().stream() ③
            .map(e -> e.getObjectName() + ":" + e.getDefaultMessage())
            .collect(Collectors.toList());
        String description = Stream.concat(fieldMsgs.stream(), globalMsgs.stream())
            .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

393.7.1. 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",
  "date" : "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.



393.8. @PathVariable Validation

Note that the Web API maps the `@RequestBody` constraint violations independently from 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

393.9. @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",
  "date" : "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"}
```

393.10. @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) {
```

393.11. @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",
  "date" : "2021-07-01T15:55:44.089734Z" }
```

① pageNumber has an invalid negative value

② pageSize has an invalid non-positive value

393.12. 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

393.13. 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) {
```

393.14. 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",
  "date":"2021-07-01T16:23:27.666154Z"}
```

393.15. 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 the 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 thru 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()
        .orElse(null)!=null;

    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 component annotated with `@RestController`

393.16. Internal Server Error Correctly Reported

The following snippet shows the response to the client when our exception handler detects that it 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",
  "date" : "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 of the time to denote classes of a certain pattern. One could create a custom annotation that explicitly indicates what we are looking to identify.

393.17. 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)
    public 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".

```
< 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",
  "date" : "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. Lets try another technique.

393.18. Payload

One other option we have to is leverage the payload metadata in each annotation. **Payload** classes are interfaces extending `javax.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) {
```

393.19. Exception Handler Checking Payloads

The snippet below shows our generic, global advice factoring in whether the violation came from an annotation with a `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()
            .orElse(null)!=null;

    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

393.20. 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",
"date" : "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",
  "date" : "2021-07-01T20:25:05.188126Z" }
```

Chapter 394. 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 27. JPA Validation Configuration Properties

<code>javax.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 provider missing• none - disable validation entirely within JPA
<code>javax.persistence.validation.group.pre-persist</code>	identify group(s) validated prior to inserting new row	<ul style="list-style-type: none">• <code>javax.validation.groups.Default.class</code> (default)
<code>javax.persistence.validation.group.pre-update</code>	identify group(s) validated prior to updating existing row	<ul style="list-style-type: none">• <code>javax.validation.groups.Default.class</code> (default)
<code>javax.persistence.validation.group.pre-remove</code>	identify group(s) validated prior to removing existing row	<ul style="list-style-type: none">• (none) (default)

Chapter 395. 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);
    }
}
```

395.1. Validating Saves

To demonstrate validation within the data tier, lets 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](#).

395.2. 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 does not take much 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.

395.3. 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 Java object and [Document](#) (e.g., save and load)
 - and after converting between Java object and [Document](#) (e.g., save and load)
 - after action is complete (e.g., save)

The following table lists the specific events.

Table 28. MongoMappingEvents

onApplicationEvent(MongoMappingEvent<?> event)	general purpose event handler
onBeforeConvert(BeforeConvertEvent<E> event)	callback before 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 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

395.4. 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 396. 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](#).

396.1. 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?

396.1.1. Data Tier Validation Safety Checks

Hombergs' suggestion was to use the data tier validation as a safety check, but not the only layer. ^[7]

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.

396.2. 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.

C Person
<code>@Null[group=Create.class] String id</code>
<code>@NotNull String name</code>

Figure 164. 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](#).

396.2.1. 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. [\[77\]](#)

One way of implementing use case-specific query methods and have 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.

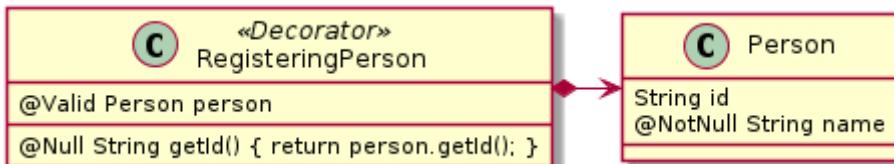


Figure 165. Use case-Specific Data Wrapper

396.3. Anti: Validation Everywhere

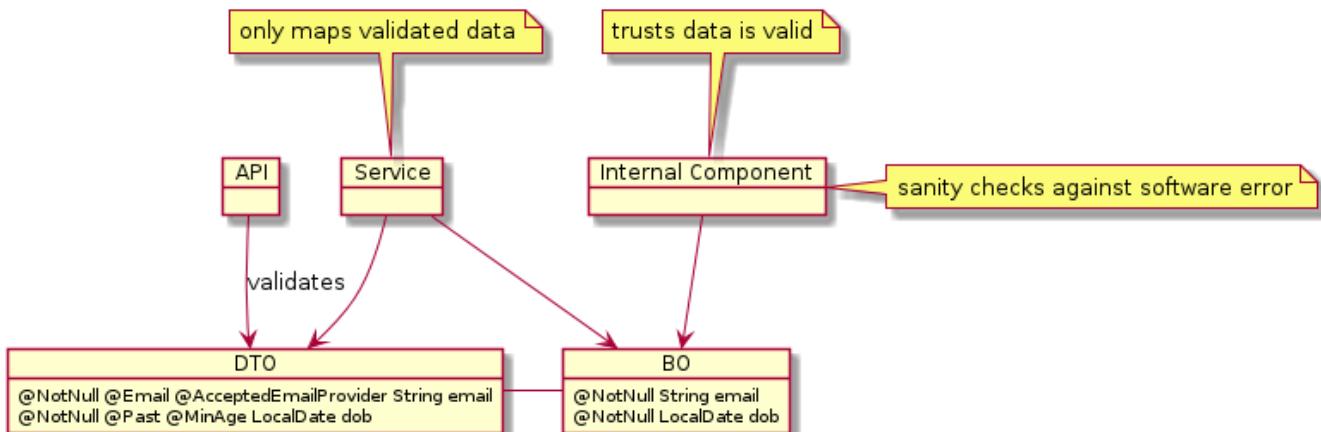
It is likely for us to want to validate at the client interface (Web API) since these are very external inputs. It is also likely for us to want to validate at the service level because our service could be injected into multiple client interfaces. It is then likely that internal components see how easy 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 it can be to determine what is a client versus internal issue when a violation occurs.

396.3.1. Establish Validation Architecture

Hombergs recommends having a clear validation strategy versus ad-hoc everywhere [\[77\]](#)

I agree with that strategy and like to have a clear dividing line of "once it reaches this point—data is 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.

396.3.2. 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.^[78] 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.

[77] "[Bean Validation Anti-Patterns](#)", Tom Hombergs

[78] "[Get Your Hands Dirty on Clean Architecture](#)", Tom Hombergs, 2019

Chapter 397. 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 identify patterns/anti-patterns for 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::/var/jenkins_home/workspace/ava_ejava-springboot-docs_master@2/courses/jhu784-notes/target/resources/docs/asciidoc/assignment6-race-async-{assignment6}.adoc[]

Integration Unit Testing

copyright © 2021 jim stafford (jim.stafford@jhu.edu)

Chapter 398. 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 "**integration unit 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 "integration unit 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...

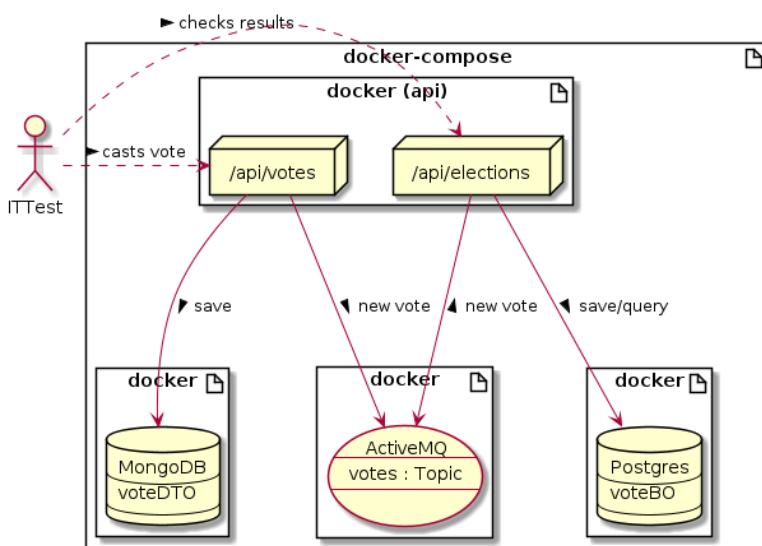


Figure 166. Votes Application

- 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?

398.1. 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 integration unit test using embedded resources

398.2. Objectives

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

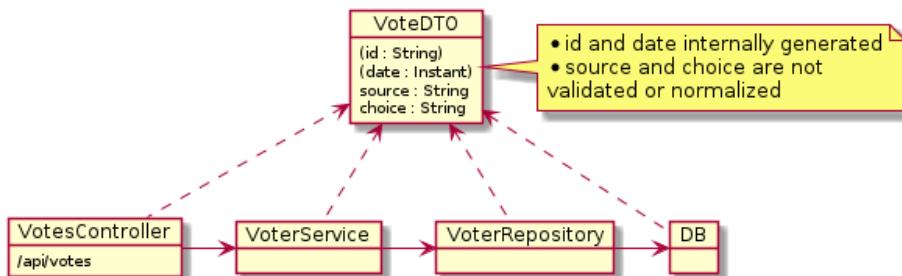
1. embed a simulated MongoDB within a JUnit test using Flapdoodle
2. embed an in-memory JMS server within a JUnit test using ActiveMQ
3. embed a relational database within a JUnit test using H2
4. verify an end-to-end test case using a unit integration test

Chapter 399. 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.

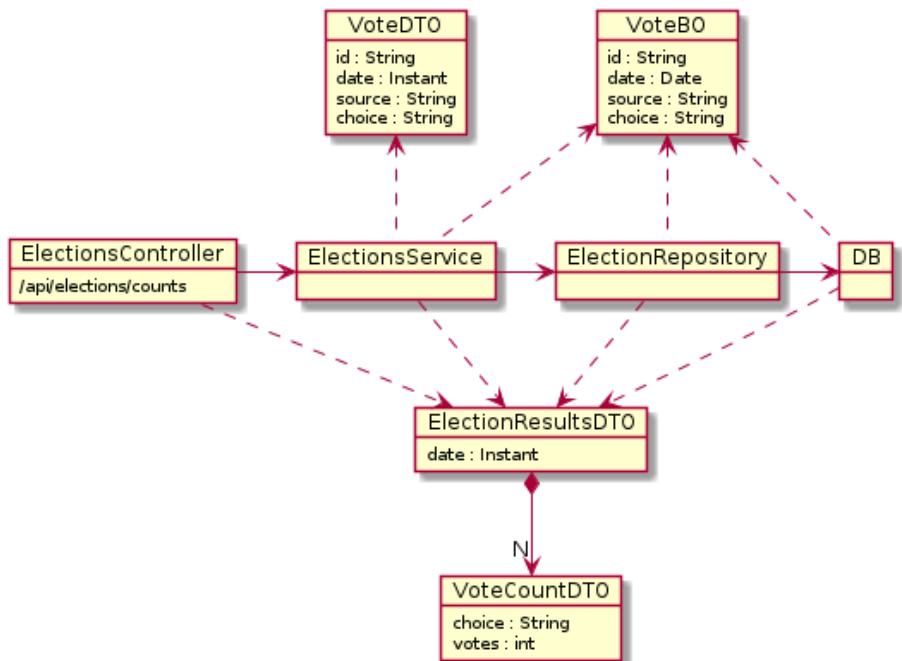
399.1. Main Application Flows

Table 29. Service Controller/Database Dependencies



The Votes Service accepts a vote (VoteDTO) from a caller and stores that directly in a database (MongoDB).

Figure 167. 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 168. 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.

399.2. 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 30. Async Dependencies

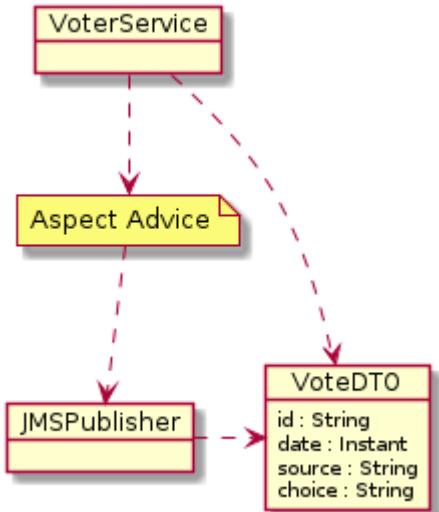


Figure 169. Votes Service

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.

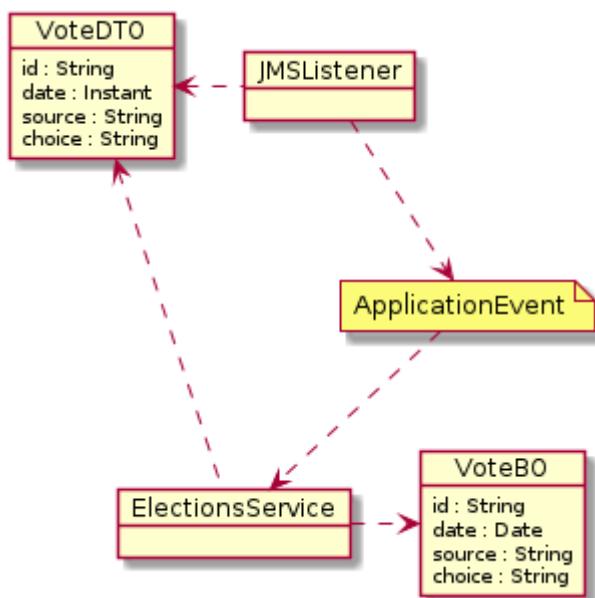


Figure 170. Elections Service

The Elections Service eventing layer subscribes to the `votes` destination and issues an internal `NewVote` POJO `ApplicationEvent`—which is relayed to the `ElectionsService` for mapping to an entity class (`VoteBO`) and storage in the DB for later query.

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.

Chapter 400. Physical Architecture

I described five (5) functional services in the previous section: Votes, Elections, MongoDB, Postgres, and ActiveMQ (for JMS).

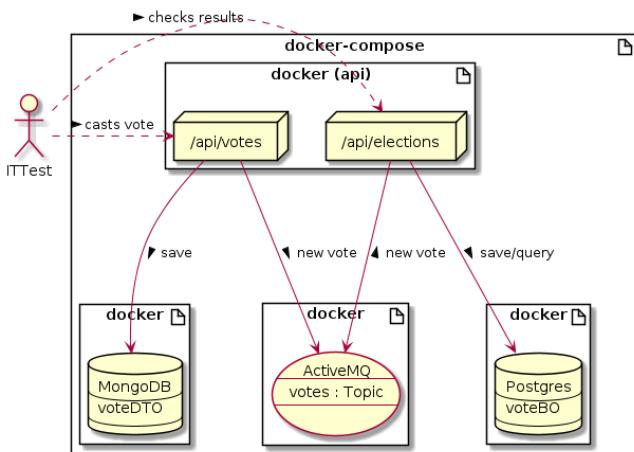


Figure 171. Physical Architecture

I will eventually mapped them to four (4) physical nodes: api, mongo, postgres, and activemq. Both Votes and Elections have been co-located in the same Spring Boot application because the Internet deployment platform may not have a JMS server available for our use.

400.1. Integration Unit Test Physical Architecture

For integration unit tests, we will use a single JUnit JVM with the Spring Boot Services and the three resources embedded using the following options:

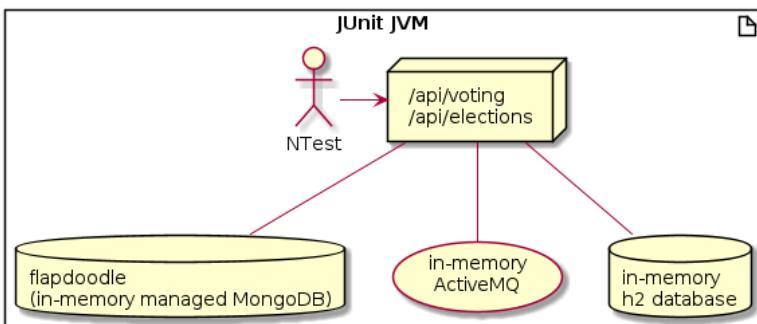


Figure 172. Integration Unit Testing Physical Architecture

- [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 401. 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.

401.1. 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:2.3.2.RELEASE:compile
[INFO] |   +- org.mongodb:mongodb-driver-sync:jar:4.0.5:compile
[INFO] |   |   +- org.mongodb:bson:jar:4.0.5:compile
[INFO] |   |   \- org.mongodb:mongodb-driver-core:jar:4.0.5:compile
[INFO] |   \- org.springframework.data:spring-data-mongodb:jar:3.0.2.RELEASE:compile
```

401.2. 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</artifactId>
    <scope>test</scope>
</dependency>
```

The [flapdoodle](#) dependency brings in the following artifacts.

Flapdoodle Dependencies

```
[INFO] +- de.flapdoodle.embed:de.flapdoodle.embed.mongo:jar:2.2.0:test
[INFO] | \- de.flapdoodle.embed:de.flapdoodle.embed.process:jar:2.1.2:test
[INFO] |   +- org.apache.commons:commons-lang3:jar:3.10:compile
[INFO] |   +- net.java.dev.jna:jna:jar:4.0.0:test
[INFO] |   +- net.java.dev.jna:jna-platform:jar:4.0.0:test
[INFO] |   \- org.apache.commons:commons-compress:jar:1.18:test
```

401.3. 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

401.4. 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.votes.dto.VoteDTO;
import org.springframework.data.mongodb.repository.MongoRepository;

public interface VoterRepository extends MongoRepository<VoteDTO, String> { }
```

401.5. 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"
}
```

401.6. 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 402. 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.

402.1. ActiveMQ Maven Dependencies

The following lists the dependencies we need to implement the Aspects and JMS capability within the application.

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-activemq</artifactId>
</dependency>
```

The ActiveMQ starter brings in the following dependencies and activates the `ActiveMQAutoConfiguration` class that will setup a JMS connection based on properties.

ActiveMQ Starter Dependencies

```
[INFO] +- org.springframework.boot:spring-boot-starter-activemq:jar:2.3.2.RELEASE:compile
[INFO] |  +- org.springframework:spring-jms:jar:5.2.8.RELEASE:compile
[INFO] |  |  +- org.springframework:spring-messaging:jar:5.2.8.RELEASE:compile
[INFO] |  |  \- org.springframework:spring-tx:jar:5.2.8.RELEASE:compile
```

402.2. ActiveMQ Integration Unit 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 traffic flow if we care to.

ActiveMQ Core Properties

```
#activemq
spring.jms.pub-sub-domain=true ①
```

① tells ActiveMQ to use topics versus queues

The following lists the properties that are unique to the local integration unit tests.

```
#activemq
spring.activemq.in-memory=true ①
spring.activemq.pool.enabled=false
```

① activemq will establish in-memory destinations

402.3. 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

402.4. 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.

```
@Component  
@RequiredArgsConstructor  
public class VoterJMS {  
    private final JmsTemplate jmsTemplate; ①  
    private final ObjectMapper mapper; ②  
    ...  
  
    public void publish(VoteDTO vote) throws JsonProcessingException {  
        final String json = mapper.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`

402.5. 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  
public Jackson2ObjectMapperBuilder jacksonBuilder() {  
    Jackson2ObjectMapperBuilder builder = new Jackson2ObjectMapperBuilder()  
        .indentOutput(true)  
        .featuresToDisable(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS);  
    return builder;  
}  
  
@Bean  
public ObjectMapper jsonMapper(Jackson2ObjectMapperBuilder builder) {  
    return builder.createXmlMapper(false).build();  
}
```

402.6. 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 mapper;

    @JmsListener(destination = "votes") ②
    public void receive(TextMessage message) throws JMSException { ①
        String json = message.getText();
        try {
            VoteDTO vote = mapper.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 unmarshaled into a `VoteDTO` instance

④ Simple `NewVote` POJO event created and issued internal

402.7. 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 403. 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.

403.1. 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.

403.2. JPA Test Dependencies

During integration unit 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>
```

403.3. 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.

JPA Test Properties

```
#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

403.4. 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 integration unit testing, we get a chance to supply a more tested product for production deployment.

403.5. 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

403.6. 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);
```

403.7. 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 1.4)
Successfully validated 2 migrations (execution time 00:00.022s)
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" (execution time 00:00.069s)
```

For our integration unit 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.

403.8. 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 JPA Query that translates into 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.jpa.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

403.9. 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 javax.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

403.10. 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 404. 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 Integration Unit Test

```
@SpringBootTest( classes = {ClientTestConfiguration.class, VotesExampleApp.class},
    webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT, ①
    properties = "test=true") ②
@ActiveProfiles("test") ③
@DisplayName("votes integration unit test")
public class VotesTemplateNTest {
    @Autowired ④
    private RestTemplate restTemplate;
    private final URI baseVotesUrl;
    private final URI baseElectionsUrl;

    public VotesTemplateNTest(@LocalServerPort int port) ①
        throws URISyntaxException {
        baseVotesUrl = new URI( ⑤
            String.format("http://localhost:%d/api/votes", port));
        baseElectionsUrl = new URI(
            String.format("http://localhost:%d/api/elections", port));
    }
    ...
}
```

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

404.1. 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;
    }
}
```

404.2. 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()).isAfterOrEqualTo(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 integration unit test with embedded resources in order to implement end-to-end testing on a small scale.

Chapter 405. 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 the 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 integration unit test MongoDB code using Flapdoodle
- how to integrate a ActiveMQ server into a Spring Boot application
 - and how to integration unit test JMS code using an embedded ActiveMQ server
- how to integrate a Postgres into a Spring Boot application
 - and how to integration unit test relational code using an in-memory H2 database
- how to implement an integration unit test using embedded resources

Docker Compose Integration Testing

copyright © 2021 jim stafford (jim.stafford@jhu.edu)

Chapter 406. Introduction

In the last lecture we looked at a set of Voting Services and configured integration unit testing using simulated or other in-memory resources to implement an end-to-end integration thread in a single process.

But what if we wanted or needed to use real resources?

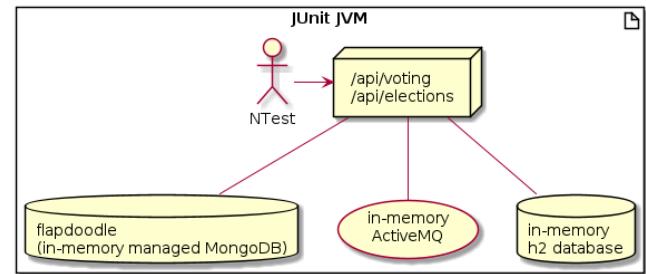


Figure 173. Integration Unit Test with In-Memory/Local Resources

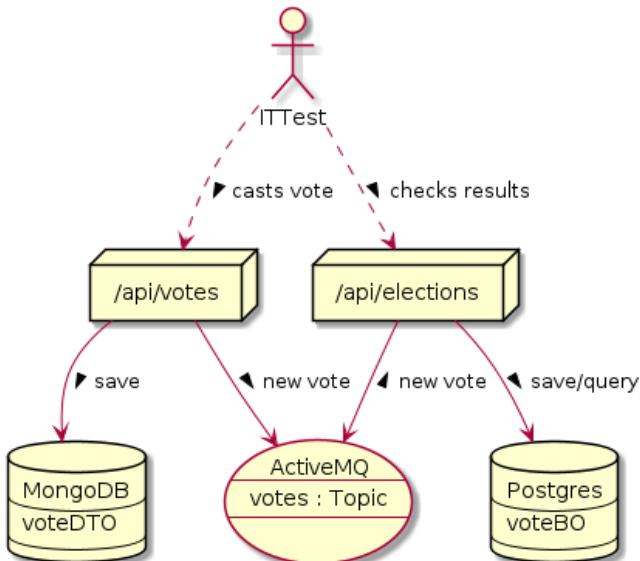


Figure 174. How Can We Test with Real Resources

What if we needed to test with a real or specific version of MongoDB, ActiveMQ, Postgres, or some other resource? What if some of those other resources were a supporting microservice?

We could implement an integration test—but how can we automate it?

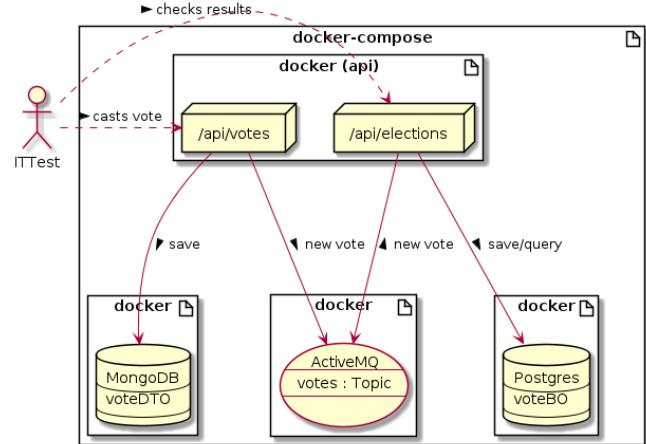


Figure 175. Integration Test with Docker and Docker Compose

406.1. Goals

You will learn:

- how to implement a network of services for development and testing using Docker Compose
- how to implement an integration test between real instances running in a virtualized environment
- how to interact with the running instances during testing

406.2. Objectives

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

1. create a Docker Compose file that defines a network of services and their dependencies
2. execute ad-hoc commands inside running images
3. integrate Docker Compose into a Maven integration test phase
4. author an integration test that uses real resource instances with dynamically assigned ports

Chapter 407. Integration Testing with Real Resources

We are in a situation where we need to run integration tests against real components. These "real" components can be virtualized, but they primarily need to contain a specific feature of a specific version we are taking advantage of.

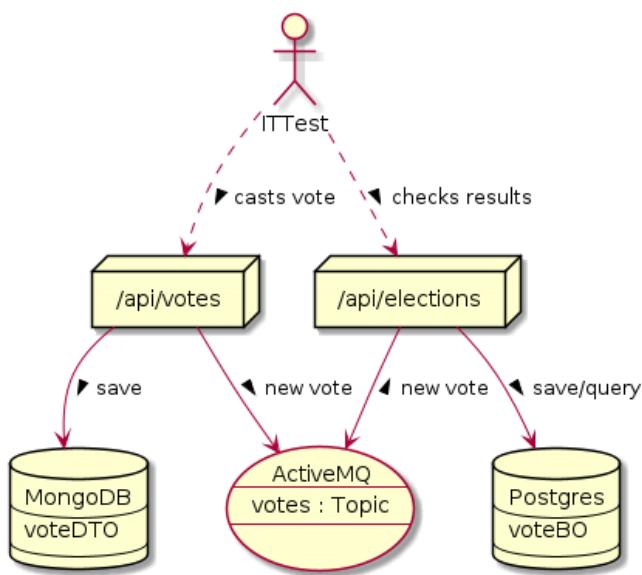


Figure 176. Need to Integrate with Specific Real Services

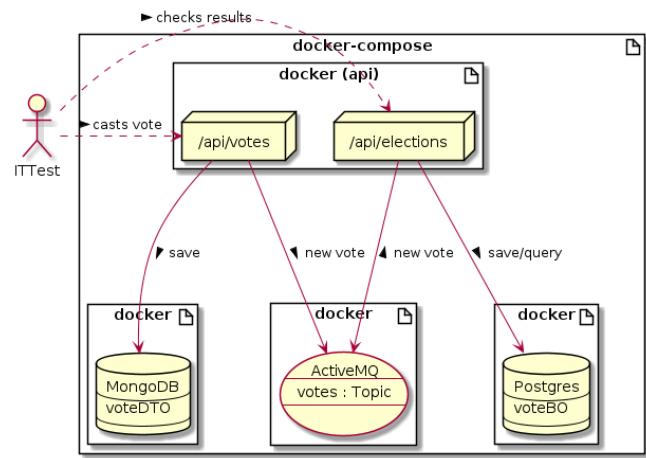


Figure 177. Virtualize Services with Docker

My example uses generic back-end resources as examples of what we need to integrate with. However, in the age of microservices—these examples could easily be lower-level applications offering necessary services for our client application to properly operate.

We need access to these resources in the development environment but would soon need them during automated integration tests running regression tests in the CI server as well.

Lets look to Docker for a solution ...

407.1. 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

$ docker run --rm -p 5432:5432 \
-e POSTGRES_PASSWORD=secret postgres:12.3-alpine

$ docker run --rm -p 61616:61616 -p 8161:8161 \
rmohr/activemq:5.15.9

$ docker run --rm -p 9080:8080 \
-e MONGODB_URI='mongodb://admin:secret@host.docker.internal:27017/votes_db?authSource=admin' \
-e DATABASE_URL='postgres://postgres:secret@host.docker.internal:5432/postgres' \
-e spring.profiles.active=integration dockercompose-votes-api:latest
```

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 machine 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 — covered earlier. I will review some of the aspects.

Chapter 408. 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 four (4) nodes, we can limit our definition to a [version](#) and [services](#).

docker-compose.yml Shell

```
version: '3.8'
services:
  mongo:
    ...
  postgres:
    ...
  activemq:
    ...
  api:
    ...
```

Refer to the [Compose File Reference](#) for more details.

408.1. 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:
#     - "27017:27017"
```

408.2. postgres Service Definition

The [postgres](#) service defines our instance of [Postgres](#).

postgres Service Definition

```
postgres:  
  image: postgres:12.3-alpine  
#  ports:  
#    - "5432:5432"  
  environment:  
    POSTGRES_PASSWORD: secret
```

408.3. activemq Service Definition

The `activemq` service defines our instance of [ActiveMQ](#).

activemq Service Definition

```
activemq:  
  image: rmohr/activemq:5.15.9  
#  ports:  
#    - "61616:61616"  
#    - "8161:8161"
```

- port 61616 is used for JMS communication
- port 8161 is an HTTP server that can be used for HTML status

408.4. api Service Definition

The `api` service defines our API server with the Votes and Elections Services. This service will become a client of the other three services.

api Service Definition

```
api:  
  build:  
    context: ../dockercompose-votes-svc  
    dockerfile: Dockerfile  
  image: dockercompose-votes-api:latest  
  ports:  
    - "${API_PORT}:8080"  
  depends_on:  
    - mongo  
    - postgres  
    - activemq  
  environment:  
    - spring.profiles.active=integration  
    - MONGODB_URI=mongodb://admin:secret@mongo:27017/votes_db?authSource=admin  
    - DATABASE_URL=postgres://postgres:secret@postgres:5432/postgres
```

408.5. Compose Override Files

I left off port definitions from the primary file on purpose. That will become more evident in the Testcontainers topic in the next lecture when we need dynamically assigned port numbers. However, for purposes here we need well-known port numbers and can do so easily with an additional configuration file—`docker-compose.override.yml`.

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.

Example Compose Override File

```
version: '3.8'  
services:  
  mongo:  
    ports:  
      - "27017:27017"  
  postgres:  
    ports:  
      - "5432:5432"  
  activemq:  
    ports:  
      - "61616:61616"  
      - "8161:8161"
```

When started—notice how the container port# is mapped according to how the override file has specified.

Port Mappings with Compose Override File Used

```
$ docker ps  
IMAGE                      PORTS  
dockercompose-votes-api:latest 0.0.0.0:9090->8080/tcp  
postgres:12.3-alpine          0.0.0.0:5432->5432/tcp, 0.0.0.0:32812->5432/tcp  
mongo:4.4.0-bionic            0.0.0.0:27017->27017/tcp, 0.0.0.0:32813->27017/tcp  
rmohr/activemq:5.15.9         1883/tcp, 5672/tcp, 0.0.0.0:8161->8161/tcp, 61613-  
61614/tcp, 0.0.0.0:61616->61616/tcp ①
```

① notice that only the ports we mapped are exposed

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.

Chapter 409. Test Drive

Lets test out our services before demonstrating a few more commands. Everything is up and running and only the API port is exposed to the local host network using port# 9090.

Running Network Port Mapping

```
$ docker ps
IMAGE                      PORTS
dockercompose-votes-api:latest 0.0.0.0:9090->8080/tcp ①
postgres:12.3-alpine          5432/tcp
mongo:4.4.0-bionic            27017/tcp
rmohr/activemq:5.15.9         1883/tcp, 5672/tcp, 8161/tcp, 61613-61614/tcp,
                               61616/tcp
```

① only the API has its container port# (8080) mapped to a host port# (9090)

409.1. Clean Starting State

We start off with nothing in the Vote or Election databases.

Clean Starting State

```
$ curl http://localhost:9090/api/votes/total
0

$ curl http://localhost:9090/api/elections/counts
{
  "date" : "1970-01-01T00:00:00Z",
  "results" : [ ]
}
```

409.2. Cast Two Votes

We can then cast votes for different choices and have them added to MongoDB and have a JMS message published.

Cast Two Votes

```
$ curl -X POST http://localhost:9090/api/votes -H "Content-Type: application/json" -d '{"source":"jim","choice":"quisp"}'  
{  
    "id" : "5f31eed580cfe474aeaa1536",  
    "date" : "2020-08-11T01:05:25.168505Z",  
    "source" : "jim",  
    "choice" : "quisp"  
}  
$ curl -X POST http://localhost:9090/api/votes -H "Content-Type: application/json" -d '{"source":"jim","choice":"quake"}'  
{  
    "id" : "5f31eee080cfe474aeaa1537",  
    "date" : "2020-08-11T01:05:36.374043Z",  
    "source" : "jim",  
    "choice" : "quake"  
}
```

409.3. Observe Updated State

At this point we can locate some election results in Postgres using API calls.

Updated State

```
$ curl http://localhost:9090/api/elections/counts  
{  
    "date" : "2020-08-11T01:05:36.374Z",  
    "results" : [ {  
        "choice" : "quake",  
        "votes" : 1  
    }, {  
        "choice" : "quisp",  
        "votes" : 1  
    } ]  
}
```

Chapter 410. Inspect Images

This is a part that I think is really useful and easy. Docker Compose provides an easy interface for running commands within the images.

410.1. Exec Mongo CLI

In the following example, I am running the `mongo` command line interface (CLI) command against the running `mongo` service and passing in credentials as command line arguments. Once inside, I can locate our `votes_db` database, `votes` collection, and two documents that represent the votes I was able to cast earlier.

Exec Command Against Running mongo Image

```
$ docker-compose exec mongo mongo -u admin -p secret --authenticationDatabase admin ①
MongoDB shell version v4.4.0
connecting to:
mongodb://127.0.0.1:27017/?authSource=admin&compressors=disabled&gssapiServiceName=mon
godb
Implicit session: session { "id" : UUID("1fb09ab-73e3-459f-b5f5-5d23903f672c") }
MongoDB server version: 4.4.0

> show dbs ②
admin      0.000GB
config     0.000GB
local      0.000GB
votes_db   0.000GB
> use votes_db
switched to db votes_db
> show collections
votes
> db.votes.find({}, {"choice":1}) ③
{ "_id" : ObjectId("5f31eed580cf474aeaa1536"), "choice" : "quisp" }
{ "_id" : ObjectId("5f31eee080cf474aeaa1537"), "choice" : "quake" }
> exit ④
bye
```

① running `mongo` CLI command inside running `mongo` image with command line args expressing credentials

② running CLI commands to inspect database

③ listing documents in the `votes` database

④ exiting CLI and returning to host shell

410.2. Exec Postgres CLI

In the following example, I am running the `psql` CLI command against the running `postgres` service and passing in credentials as command line arguments. Once inside, I can locate our Flyway

migration and VOTE table and list some of the votes that are in the election.

Exec Command Against Running postgres Image

```
$ docker-compose exec postgres psql -U postgres ①
psql (12.3)
Type "help" for help.

postgres=# \d+ ②
                                         List of relations
 Schema |        Name         | Type  | Owner   |     Size    |          Description
-----+-----+-----+-----+-----+
 public | flyway_schema_history | table | postgres | 16 kB
 public | vote                 | table | postgres | 8192 bytes | countable votes for
election
(2 rows)

postgres=# select * from vote; ③
      id       | choice |           date           | source
-----+-----+-----+-----+-----+
 5f31eed580cfe474aeaa1536 | quisp  | 2020-08-11 01:05:25.168 | jim
 5f31eee080cfe474aeaa1537 | quake   | 2020-08-11 01:05:36.374 | jim
(2 rows)

postgres=# \q ④
```

① running `psql` CLI command inside running `postgres` image with command line args expressing credentials

② running CLI commands to inspect database

③ listing table rows in the vote table

④ exiting CLI and returning to host shell

410.3. Exec Impact

With the capability to exec a command inside the running containers, we can gain access to a significant amount of state of our application and databases without having to install any software beyond Docker.

Chapter 411. Integration Test Setup

At this point we should understand what Docker Compose is and how to configure it for use with our specific integration test. I now want to demonstrate it being used in an automated "integration test" where it will get executed as part of the Maven integration-test phases.

411.1. Integration Properties

We will be launching our API image with the following Docker environment expressed in the Docker Compose file.

API Docker Environment

```
environment:  
  - spring.profiles.active=integration  
  - MONGODB_URI=mongodb://admin:secret@mongo:27017/votes_db?authSource=admin  
  - DATABASE_URL=postgres://postgres:secret@postgres:5432/postgres
```

That will get digested by the run_env.sh script to produce the following.

Spring Boot Properties

```
--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
```

That will be integrated with the following properties from the `integration` profile.

application-integration.properties

```
#activemq  
spring.activemq.broker-url=tcp://activemq:61616  
  
#rdbms  
spring.jpa.show-sql=true  
spring.jpa.generate-ddl=false  
spring.jpa.hibernate.ddl-auto=validate  
spring.flyway.enabled=true
```

I have chosen to hard-code the integration URL for ActiveMQ into the properties file since we won't be passing in an ActiveMQ URL in production. The MongoDB and Postgres properties will originate from environment variables versus hard coding them into the integration properties file to better match the production environment and further test the run_env.sh launch script.

411.2. Maven Build Helper Plugin

We will want a random, not in use port# assigned when we run the integration tests so that multiple instances of the test can be run concurrently on the same build server without colliding. We can leverage the [build-helper-maven-plugin](#) to identify a port# and have it assigned the value to a Maven property. I am assigning it to a `docker.http.port` property that I made up.

Generate Random Port# for Integration Test

```
<!-- assigns a random port# to property server.http.port -->
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>build-helper-maven-plugin</artifactId>
    <executions>
        <execution>
            <id>reserve-network-port</id>
            <goals>
                <goal>reserve-network-port</goal>
            </goals>
            <phase>pre-integration-test</phase>
            <configuration>
                <portNames>
                    <portName>docker.http.port</portName> ①
                </portNames>
            </configuration>
        </execution>
    </executions>
</plugin>
```

① a dynamically obtained network port# is assigned to the `docker.http.port` Maven property

The following is an example output of the [build-helper-maven-plugin](#) during the build.

Example Maven Build Helper Plugin Output

```
[INFO] --- build-helper-maven-plugin:3.1.0:reserve-network-port (reserve-network-port)
@ dockercompose-votes-it ---
[INFO] Reserved port 60616 for docker.http.port
```

411.3. Maven Docker Compose Plugin

After generating a random port#, we can start our Docker Compose network. I am using the <https://github.com/br4chu/docker-compose-maven-plugin> `docker-compose-maven-plugin`] to perform that role. It automatically hooks into the `pre-integration-test` phase to issue the `up` command and the `post-integration-test` phase to issue the `down` command when we configure it the following way. It also allows us to name and pass variables into the Docker Compose file.

```

<plugin>
  <groupId>io.brachu</groupId>
  <artifactId>docker-compose-maven-plugin</artifactId>
  <configuration>
    < projectName>${project.artifactId}</projectName>
    < file>${project.basedir}/docker-compose.yml</file>
    <env>
      <API_PORT>${docker.http.port}</API_PORT> ①
    </env>
  </configuration>
  <executions>
    <execution>
      <goals>
        <goal>up</goal>
        <goal>down</goal>
      </goals>
    </execution>
  </executions>
</plugin>

```

① dynamically obtained network port# is assigned to Docker Compose file's `API_PORT` variable, which controls the port mapping of the API server

411.4. Maven Docker Compose Plugin Output

The following shows example plugin output during the `pre-integration-test` phase that is starting the services prior to running the tests.

Example Maven Docker Compose Plugin pre-integration-test Output

```

[INFO] --- docker-compose-maven-plugin:0.8.0:up (default) @ dockercompose-votes-it ---
Creating network "dockercompose-votes-it_default" with the default driver
...
Creating dockercompose-votes-it_mongo_1    ... done
Creating dockercompose-votes-it_api_1     ... done

```

The following shows example plugin output during the `post-integration-test` phase that is shutting down the services after running the tests.

Example Maven Docker Compose Plugin post-integration-test Output

```
[INFO] --- docker-compose-maven-plugin:0.8.0:down (default) @ dockercompose-votes-it
---
Killing dockercompose-votes-it_api_1      ...
Killing dockercompose-votes-it_api_1      ... done
Killing dockercompose-votes-it_postgres_1 ... done
Removing dockercompose-votes-it_mongo_1   ... done
Removing dockercompose-votes-it_postgres_1 ... done
Removing network dockercompose-votes-it_default
```

411.5. Maven Failsafe Plugin

The following shows the configuration of the `maven-failsafe-plugin`. Generically, it runs in the `integration-test` phase, matches/runs the `IT` tests, and adds test classes to the classpath. More specific to Docker Compose—it accepts the dynamically assigned port# and passes it to JUnit using the `it.server.port` property.

Example Failsafe Plugin Configuration

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-failsafe-plugin</artifactId>
  <executions>
    <execution>
      <id>integration-test</id>
      <goals>
        <goal>integration-test</goal>
      </goals>
      <configuration>
        <includes>
          <include>**/*IT.java</include>
        </includes>
        <systemPropertyVariables>
          <it.server.port>${docker.http.port}</it.server.port> ①
        </systemPropertyVariables>
        <additionalClasspathElements>
          <additionalClasspathElement>${basedir}/target/classes</additionalClasspathElement>
        </additionalClasspathElements>
      </configuration>
    </execution>
  </executions>
</plugin>
```

① passing in generated `docker.http.port` value into `it.server.port` property

At this point, both Docker Compose and Failsafe/JUnit have been given the same dynamically assigned port#.

411.6. IT Test Client Configuration

The following shows the IT test configuration class that maps the `it.server.port` property to the `baseUrl` for the tests.

ClientTestConfiguration Mapping it.server.port to baseUrl

```
@SpringBootConfiguration()
@EnableAutoConfiguration      //needed to setup logging
public class ClientTestConfiguration {
    @Value("${it.server.host:localhost}")
    private String host;
    @Value("${it.server.port:9090}") ①
    private int port;

    @Bean
    public URI baseUrl() {
        return UriComponentsBuilder.newInstance()
            .scheme("http")
            .host(host)
            .port(port)
            .build()
            .toUri();
    }
    @Bean
    public URI electionsUrl(URI baseUrl) {
        return UriComponentsBuilder.fromUri(baseUrl).path("api/elections")
            .build().toUri();
    }
    @Bean
    public RestTemplate anonymousUser(RestTemplateBuilder builder) {
        RestTemplate restTemplate = builder.build();
        return restTemplate;
    }
}
```

① API port# property injected through Failsafe plugin configuration

411.7. Example Failsafe Output

The following shows the Failsafe and JUnit output that runs during the `integration-test`.

Example Failsafe Output

```
[INFO] --- maven-failsafe-plugin:3.0.0-M4:integration-test (integration-test) @
dockercompose-votes-it ---
[INFO]
[INFO] -----
[INFO] T E S T S
[INFO] -----
[INFO] Running info.ejava.examples.svc.docker.votes.ElectionIT
...
...ElectionIT#init:46 votesUrl=http://localhost:60616/api/votes ①
...ElectionIT#init:47 electionsUrl=http://localhost:60616/api/elections
...
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 10.372 s - in
info.ejava.examples.svc.docker.votes.ElectionIT
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

① URLs with dynamic host port# assigned for API

411.8. IT Test Setup

The following shows the common IT test setup where the various URLs are being constructed around the injected.

IT Test Setup

```
@SpringBootTest(classes={ClientTestConfiguration.class},
    webEnvironment = SpringBootTest.WebEnvironment.NONE)
@Slf4j
public class ElectionIT {
    @Autowired
    private RestTemplate restTemplate;
    @Autowired
    private URI votesUrl;
    @Autowired
    private URI electionsUrl;
    private static Boolean serviceAvailable;

    @PostConstruct
    public void init() {
        log.info("votesUrl={}, votesUrl");
        log.info("electionsUrl={}, electionsUrl");
    }
}
```

411.9. Wait For Services Startup

We have at least one more job to do before our tests—we have to wait for the API server to finish starting up. We can add that logic to a @BeforeEach and remember the answer from the first attempt in all following attempts.

Example Wait For Services Startup

```
@BeforeEach
public void serverRunning() {
    List<URI> urls = new ArrayList<>(Arrays.asList(
        UriComponentsBuilder.fromUri(votesUrl).path("/total").build().toUri(),
        UriComponentsBuilder.fromUri(electionsUrl).path("/counts").build().toUri()
    ));

    if (serviceAvailable!=null) { assumeTrue(serviceAvailable);}
    else {
        assumeTrue(() -> { ①
            for (int i=0; i<10; i++) {
                try {
                    for (Iterator<URI> itr = urls.iterator(); itr.hasNext();) {
                        URI url = itr.next();
                        restTemplate.getForObject(url, String.class); ②
                        itr.remove(); ③
                    }
                    return serviceAvailable = true; ④
                } catch (Exception ex) {
                    //...
                }
            }
            return serviceAvailable=false;
        });
    }
}
```

① Assume.assertTrue will not run the tests if evaluates false

② checking for a non-exception result

③ removing criteria once satisfied

④ evaluate true if all criteria satisfied

At this point our tests are the same as most other Web API test where we invoke the server using HTTP calls using the assembled URLs.

Chapter 412. Summary

In this module we learned:

- to create a Docker Compose file that defines a network of services and their dependencies
- to integrate Docker Compose into a Maven integration test phase
- to implement an integration test that uses dynamically assigned ports
- execute ad-hoc commands inside running images

Testcontainers

copyright © 2021 jim stafford (jim.stafford@jhu.edu)

Chapter 413. Introduction

In a previous section we implemented "unit integration tests" with in-memory instances for back-end resources. We later leveraged Docker and Docker Compose to implement "integration tests" with real resources operating in a virtual environment. We self-integrated Docker Compose in that later step, using several Maven plugins and Maven's integration testing phases.

In this lecture I will demonstrate an easier, more seamless way to integrate Docker Compose 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.

413.1. Goals

You will learn:

- how to better integrate Docker and DockerCompose into unit tests
- how to inject dynamically assigned values into the application context startup

413.2. Objectives

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

1. implement an integration unit test using Docker Compose and Testcontainers library
2. implement a [Spring DynamicPropertySource](#) to obtain dynamically assigned port numbers in time for concrete URL injections
3. execute shell commands from a JUnit test into a running Docker container using Testcontainers library
4. establish client connection to back-end resources to inspect state as part of the test

Chapter 414. Testcontainers Overview

[Testcontainers](#) is a Java library that supports running Docker containers within JUnit tests and other test frameworks.

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.

Spring making changes to support Testcontainers



As a self observation—by looking at documentation, articles, and timing of feature releases—it is my opinion that Spring and Spring Boot are very high on Testcontainers and have added features to their framework to help make testing with Testcontainers as seamless as possible.

Chapter 415. Example

This example builds on the previous Docker Compose lecture that uses the same Votes and Elections services. The main difference is that we will be directly interfacing with the Docker images using Testcontainers in the `test` phase versus starting up the resources at the beginning of the tests and shutting down at the end.

By having such direct connect with the containers—we can control what gets reused from test to test. Sharing reused container state between tests can be error prone. Starting up and shutting down containers takes a noticeable amount of time to complete. Alternatively, we want to have more control over when we do which approach without going through extreme heroics.

415.1. Maven Dependencies

The following lists the Testcontainers Maven dependencies. The core library calls are within the `testcontainers` artifact and JUnit-specific capabilities are within the `junit-jupiter` artifact. I have declared `junit-jupiter` dependency at the `test` scope and `testcontainers` at `compile` (default) scope because

- this is a pure test module—with no packaged implementation code
- helper methods have been placed in `src/main`
- as the test suite grows larger, this allows the helper code and other test support features to be shared among different testing modules

Testcontainers Maven Dependencies

```
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>testcontainers</artifactId> ①
</dependency>
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>junit-jupiter</artifactId> ②
    <scope>test</scope>
</dependency>
```

① core Testcontainers calls will be placed in `src/main` to begin to form a test helper library

② JUnit-specific calls will be placed in `src/test`

415.2. Main Tree

The module's main tree contains a source copy of the Docker Compose file describing the network of services, a helper class that encapsulates initialization and configuration status of the network, and a JMS listener that can be used to subscribe to the JMS messages between the Voters and Elections services.

Module Main Tree

```
src/main/
|-- java
|   '-- info
|     '-- ejava
|       '-- examples
|         '-- svc
|           '-- docker
|             '-- votes
|               |-- ClientTestConfiguration.java
|               '-- VoterListener.java
|-- resources
  '-- docker-compose-votes.yml
```

415.3. Test Tree

The test tree contains artifacts that are going to pertain to this test only. The JUnit test will rely heavily on the artifacts in the `src/main` tree and we should try to work like that might come in from a library shared by multiple integration unit tests.

Module Test Tree

```
src/test/
|-- java
|   '-- info
|     '-- ejava
|       '-- examples
|         '-- svc
|           '-- docker
|             '-- votes
|               '-- ElectionCCTest.java
|-- resources
|   |-- application.properties
|   '-- junit-platform.properties
```

Chapter 416. Example: Main Tree Artifacts

The main tree contains artifacts that are generic to serving up the network for specific tests hosted in the `src/test` tree. This division has nothing directly related to do with Testcontainers — except to show that once we get one of these going, we are going to want more.

416.1. Docker Compose File

Our Docker Compose file is tucked away within the test module since it is primarily meant to support testing. I have purposely removed all external port mapping references because they are not needed. Testcontainers will provide another way to map and locate the host port#. I have eliminated the build of the image. It should have been built by now based on Maven module dependencies. However, if we can create a resolvable source reference to the module — Testcontainers will make sure it is built.

Docker Compose File For Test

```
version: '3.8'
services:
  mongo:
    image: mongo:4.4.0-bionic
    environment:
      MONGO_INITDB_ROOT_USERNAME: admin
      MONGO_INITDB_ROOT_PASSWORD: secret
  postgres:
    image: postgres:12.3-alpine
    environment:
      POSTGRES_PASSWORD: secret
  activemq:
    image: rmohr/activemq:5.15.9
  api:
    image: dockercompose-votes-api:latest
    depends_on:
      - mongo
      - postgres
      - activemq
    environment:
      - spring.profiles.active=integration
      - MONGODB_URI=mongodb://admin:secret@mongo:27017/votes_db?authSource=admin
      - DATABASE_URL=postgres://postgres:secret@postgres:5432/postgres
```

416.2. Docker Compose File Reference

Testcontainers will load one to many layered Docker Compose files — but insists that they each be expressed as a `java.io.File`. If we assume the code in the `src/main` tree is always going to be in source form — then we can make a direct reference there. However, assuming that this could be coming from a JAR — I decided to copy the data from classpath and into a referencable file in the target tree.

Obtaining Portable File Reference from Classpath

```
import java.io.File;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardCopyOption;
...
public static File composeFile() {
    Path targetPath = Paths.get("target/docker-compose-votes.yml"); ②
    try (InputStream is = ClientTestConfiguration.class ①
        .getResourceAsStream("/docker-compose-votes.yml")) {
        Files.copy(is, targetPath, StandardCopyOption.REPLACE_EXISTING);
    } catch (IOException ex) {
        Assertions.fail("error creating source Docker Compose file", ex);
    }
    return targetPath.toFile();
}
```

① assuming worse case that the file will be coming in from a test support JAR

② placing referencable file in target path — actual name does not matter

The following shows the source and target locations of the Docker Compose file written out.

Writing Out Docker Compose File

```
target/
| '-- classes/
|   '-- docker-compose-votes.yml ①
`-- docker-compose-votes.yml ②
```

① source coming from classpath

② target written as a known file in target directory

416.3. DockerComposeContainer

Testcontainers provides [many containers](#) — including a generic Docker container, image-specific containers, and a Docker Compose container. We are going to leverage our knowledge of Docker Compose and the encapsulation of details of the Docker Compose file here and have Testcontainers directly parse the Docker Compose file.

The example shows us supplying a project name, file reference(s), and then exposing individual container ports from each of the services. Originally — only the API port needed to be exposed. However, because of the simplicity to do more with Testcontainers, I am going to expose the other ports as well. Testcontainers will also conveniently wait for activity on each of the ports when the network is started — before returning control back to our test. This can eliminate the need for "is server ready?" checks.

```

public static DockerComposeContainer testEnvironment() {
    DockerComposeContainer env =
        new DockerComposeContainer("testcontainers-votes", composeFile())
            .withExposedService("api", 8080) ①
            .withExposedService("activemq", 61616) ②
            .withExposedService("postgres", 5432) ②
            .withExposedService("mongo", 27017) ②
            .withLocalCompose(true); ③
    return env;
}

```

- ① exposing container ports using random port and will wait for container port to become active
- ② optionally exposing lower level resource services to demonstrate further capability
- ③ indicates whether this is a host machine that will run the images as children or whether this is running as a Docker image and the images will be tunneled ([wormholed](#)) out as sibling containers

416.4. Obtaining Runtime Port Numbers

At runtime, we can obtain the assigned hostname and port numbers by calling [getServiceHost\(\)](#) and [getServicePort\(\)](#) with the service name and container port we exposed earlier.

Obtaining Runtime Port Numbers

```

DockerComposeContainer env = ClientTestConfiguration.testEnvironment(); ①
...
env.start(); ②

env.getServicePort("api", 8080); ③
env.getServiceHost("mongo", null); ④
env.getServicePort("mongo", 27017);
env.getServiceHost("activemq", null);
env.getServicePort("activemq", 61616);
env.getServiceHost("postgres", null);
env.getServicePort("postgres", 5432);

```

- ① Docker Compose file is parsed
- ② network/services must be started in order to determine mapped host port numbers
- ③ referenced port must have been listed with [withExposedService\(\)](#) earlier
- ④ hostname is available as well if ever not available on [localhost](#). Second param not used.

Chapter 417. Example: Test Tree Artifacts

417.1. Primary NTest Setup

We construct our test as a normal Spring Boot integration unit test (NTest) except we have no core application to include in the Spring context—everything is provided through the test configuration. There is no need for a web server—we will use HTTP calls from the test's JVM to speak to the remote web server.

Docker images and Docker Compose networks of services take many seconds (~10-15secs) to completely startup. Thus we want to promote some level of efficiency between tests. We will instantiate and store the `DockerComposeContainer` in a static variable, initialize and shutdown once per test class, and reuse for each test method within that class. Since we are sharing the same network for each test method—I am also demonstrating the ability to control the order of the test methods.

Lastly—we can have the lifecycle of the network integrated with the JUnit test case by adding the `@Testcontainers` annotation to the class and the `@Container` annotation to the field holding the overall container. This takes care of automatically starting and stopping the network defined in the `env` variable.

Primary NTest Setup

```
import org.testcontainers.containers.DockerComposeContainer;
import org.testcontainers.junit.jupiter.Container;
import org.testcontainers.junit.jupiter.Testcontainers;
...
@Testcontainers(⑤)
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)(④)
@SpringBootTest(classes={ClientTestConfiguration.class}, ①
    webEnvironment = SpringBootTest.WebEnvironment.NONE) ②
public class ElectionCCTest {
    @Container(⑤)
    private static DockerComposeContainer env = ③
        ClientTestConfiguration.testEnvironment();

    @Test @Order(1)
    public void vote_counted_in_election() { //...
    @Test
    @Order(3) ④
    public void test3() { vote_counted_in_election(); }
    @Test @Order(2)
    public void test2() { vote_counted_in_election(); }
```

① Only test constructs in our application context—no application beans

② we do not need a web server—we are the client of a web server

③ sharing same network in all tests within this test case

- ④ controlling order of tests when using shared network
- ⑤ `@Testcontainers` and `@Container` annotations integrate the lifecycle of the network with the test case

417.2. Injecting Dynamically Assigned Port#s

We soon hit a chicken-and-the-egg problem when we attempt to inject the URLs int the test class.

```
@Autowired
private URI votesUrl;
@Autowired
private URI electionsUrl;
```

- The test class attempts to `@Autowire` URLs for the services

- the `@Bean` factories build the URLs from the host and port number

```
@Bean
public URI baseUrl() {
    return UriComponentsBuilder
.newInstance()
.host(host)
.port(port) // ...
@Bean
public URI votesUrl(URI baseUrl) { // ...
@Bean
public URI electionsUrl(URI baseUrl)
{ // ...
```

- the host and port number are injected into the configuration class using values from the Spring context

```
@SpringBootConfiguration()
public class ClientTestConfiguration {
    @Value("${it.server.host:localhost}")
    private String host;
    @Value("${it.server.port:9090}")
    private int port;
```

```
@Container
private static DockerComposeContainer env =
ClientTestConfiguration.testEnvironment
();
// --
@Autowired
private URI votesUrl;
@Autowired
private URI electionsUrl;
```

- the port number information is not available until after the network is started and the network is not started until just before the first test

417.3. DynamicPropertySource

In what seemed like a special favor to Testcontainers—Spring added a `DynamicPropertySource` construct to the framework that allows for a property to be supplied late in the startup process.

- after starting the network but prior to injecting any URIs and running a test, Spring invokes the following annotated method in the JUnit test so that it may inject any late properties.

```
@DynamicPropertySource  
private static void properties(DynamicPropertyRegistry registry) { ①  
    ClientTestConfiguration.initProperties(registry, env);  
}
```

① method is required to be static

- the callback method can then supply the missing property that will allow for the URI injections needed for the tests

```
public static void initProperties(DynamicPropertyRegistry registry,  
DockerComposeContainer env){  
    registry.add("it.server.port", ()->env.getServicePort("api", 8080));  
    //...  
}
```

Nice!

417.4. Injections Complete prior to Tests

With the injections in place, we can show that URLs with the dynamically assigned port numbers. We also have the opportunity to have the test wait for anything we can think of. Testcontainers waited for the container port to become active. The example below instructs Testcontainers to wait for our API calls to be available as well. This eliminates the need for that ugly `@BeforeEach` call in the last lecture where we needed to wait for the API server to be ready before running the tests.

Example @BeforeEach

```
@BeforeEach  
public void init() throws IOException, InterruptedException {  
    log.info("votesUrl={}", votesUrl); ①  
    log.info("electionsUrl={}", electionsUrl);  
  
    /**  
     * wait for various events relative to our containers  
     */  
    env.waitingFor("api", Wait.forHttp(votesUrl.toString())); ②  
    env.waitingFor("api", Wait.forHttp(electionsUrl.toString()));
```

① logging injected URLs with dynamically assigned host port numbers

② instructing Testcontainers to also wait for the API to come available

Example URLs with Dynamically Assigned Port Numbers

```
ElectionCTest#init:73 votesUrl=http://localhost:32989/api/votes
```

```
ElectionCTest#init:74 electionsUrl=http://localhost:32989/api/votes
```

Chapter 418. Exec Commands

Testcontainers gives us the ability to execute commands against specific running containers. The following executes the database CLI interfaces, requests a dump of information, and then obtains the results from stdout.

Example Commands Issued to Running Containers

```
import org.testcontainers.containers.Container.ExecResult;
import org.testcontainers.containers.ContainerState;
...
ContainerState mongo = (ContainerState) env.getContainerByServiceName("mongo_1")
    .orElseThrow();
ExecResult result = mongo.execInContainer("mongo",
    "-u", "admin", "-p", "secret", "--authenticationDatabase", "admin",
    "--eval", "db.getSiblingDB('votes_db').votes.find()");
log.info("voter votes = {}", result.getStdout());

ContainerState postgres = (ContainerState)env.getContainerByServiceName("postgres_1")
    .orElseThrow();
result = postgres.execInContainer("psql",
    "-U", "postgres",
    "-c", "select * from vote");
log.info("election votes = {}", result.getStdout());
```

That is a bit unwieldy, but demonstrates what we can do from a shell perspective and we will improve on this in a moment by using the API.

418.1. Exec MongoDB Command Output

The following shows the stdout obtained from the MongoDB container after executing the login and query of the `votes` collection.

Exec MongoDB Command Output

```
ElectionCTest#init:105 voter votes = MongoDB shell version v4.4.0
connecting to:
mongodb://127.0.0.1:27017/?authSource=admin&compressors=disabled&gssapiServiceName=mon
godb
Implicit session: session { "id" : UUID("5f903fe7-b43c-4ce8-b6ae-7ef53fcfb434") }
MongoDB server version: 4.4.0
{ "_id" : ObjectId("5f357fef01737362e202a96d"), "date" : ISODate("2020-08-
13T18:01:19.872Z"), "source" : "b67e012e-3e2f-4a66-b24b-b64d06d9b4c2", "choice" :
"quisp-de5fd4f2-8ab8-4997-852e-2fb97862c87", "_class" :
"info.ejava.examples.svc.docker.votes.dto.VoteDTO" }
{ "_id" : ObjectId("5f357ff001737362e202a96e"), "date" : ISODate("2020-08-
13T18:01:20.515Z"), "source" : "af366d9b-53cb-4487-8f21-e634eca08d67", "choice" :
"quake-784f3df6-c6c4-4c3b-8d45-58636b335096", "_class" :
"info.ejava.examples.svc.docker.votes.dto.VoteDTO" }
...
...
```

418.2. Exec Postgres Command Output

The following shows the stdout from the Postgres container after executing the login and query of the `VOTE` table.

Exec Postgres Command Output

```
ElectionCTest#init:99 election votes =
      id          |           choice          |       date
      |           source
-----+-----+
-----+-----+
 5f357fef01737362e202a96d | quisp-de5fd4f2-8ab8-4997-852e-2fb97862c87 | 2020-08-13
18:01:19.872 | b67e012e-3e2f-4a66-b24b-b64d06d9b4c2
 5f357ff001737362e202a96e | quake-784f3df6-c6c4-4c3b-8d45-58636b335096 | 2020-08-13
18:01:20.515 | af366d9b-53cb-4487-8f21-e634eca08d67
...
(6 rows)
```

Chapter 419. Connect to Resources

Executing a command against a running service may be useful for interactive work. In fact, we could create a breakpoint in the test and then manually go out to inspect the back-end resources (using `docker ps` to locate the container and `docker exec` to run a shell within the container) if we have access to the host network.

However, it can be clumsy to make any sense of the stdout result when writing an automated test. If we actually need to get state from the resource—it will be much simpler to use a first-class resource API to obtain results.

Lets do that now.

419.1. Maven Dependencies

To add resource clients for our three back-end resources we just need to add the following familiar dependencies. We first introduced them in the API module's dependencies in an earlier lecture.

Back-end Resource Connection Dependencies

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-activemq</artifactId>
</dependency>
```

419.2. Injected Clients

Resource Clients to be Injected

The following resource clients will be injected into the test class. These are made available by the individual AutoConfiguration class for the resource types.

```
@Autowired  
private MongoClient mongoClient;  
@Autowired  
private JmsTemplate jmsTemplate;  
@Autowired  
private JdbcTemplate jdbcTemplate;
```

Required URL properties

The AutoConfiguration classes will require the following properties defined

```
spring.data.mongodb.uri  
spring.activemq.broker-url  
spring.datasource.url
```

419.3. URL Templates

The URLs can be built using the following hard-coded helper methods as long as we know the host and port number of each service.

URL Template Helper Methods

```
public static String mongoUrl(String host, int port) {  
    return String.format("mongodb://admin:secret@%s:%d/votes_db?authSource=admin",  
host, port);  
}  
public static String jmsUrl(String host, int port) {  
    return String.format("tcp://%s:%s", host, port);  
}  
public static String jdbcUrl(String host, int port) {  
    return String.format("jdbc:postgresql://%s:%d/postgres", host, port);  
}
```

419.4. Providing Dynamic Resource URL Declarations

The host and port numbers can be supplied from the network—just like we did with the API. Therefore, we can expand the dynamic property definition to include the three other properties.

Dynamic Property Definitions

```
public static void initProperties(DynamicPropertyRegistry registry,
DockerComposeContainer env) {
    registry.add("it.server.port", ()->env.getServicePort("api", 8080));
    registry.add("spring.data.mongodb.uri", ()-> mongoUrl( ②
        env.getServiceHost("mongo", null),
        env.getServicePort("mongo", 27017))); ①
    registry.add("spring.activemq.broker-url", ()->jmsUrl(
        env.getServiceHost("activemq", null),
        env.getServicePort("activemq", 61616)));
    registry.add("spring.datasource.url", ()->jdbcUrl(
        env.getServiceHost("postgres", null),
        env.getServicePort("postgres", 5432)));
}
```

① dynamically assigned host port numbers are made available from running network

② properties are provided to Spring late in the startup process—but in time to inject before the tests

419.5. Application Properties

The dynamically created URLs properties will be joined up with the following hard-coded application properties to complete and connection information.

Hard-coded Application Properties

```
#activemq
spring.jms.pub-sub-domain=true

#postgres
spring.datasource.driver-class-name=org.postgresql.Driver
spring.datasource.username=postgres
spring.datasource.password=secret
```

419.6. JMS Listener

To obtain the published JMS messages—we add the following component with a JMS Listener method. This will print a debug of the message and increment a counter.

```

// ...
import org.springframework.jms.annotation.JmsListener;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.TextMessage;

@Component
@Slf4j
public class VoterListener {
    @Getter
    private AtomicInteger msgCount=new AtomicInteger(0);

    @JmsListener(destination = "votes")
    public void receive(Message msg) throws JMSException {
        log.info("jmsMsg={}, {}", msgCount.incrementAndGet(), ((TextMessage) msg)
.getText());
    }
}

```

We must add the JMS listener class to the Spring application context of the test. The following example shows that being explicitly done in the `@SpringBootTest.classes` annotation.

Add Component to Test Application Context

```

@SpringBootTest(classes={ClientTestConfiguration.class, VoterListener.class}, ①
    webEnvironment = SpringBootTest.WebEnvironment.NONE)
//...
public class ElectionCNTTest {

```

① adding VoterListener component class to Spring context

419.7. Obtain Client Status

The following shows a set calls to the client interfaces to show the basic capability to communicate with the network services. This gives us the ability to add debug or obscure test verification.

Example Network Service Client Calls

```
@BeforeEach
public void init() throws IOException, InterruptedException {
    ...
    /**
     * connect directly to exposed port# of images to obtain sample status
     */
    log.info("mongo client vote count={}", ①
        mongoClient.getDatabase("votes_db").getCollection("votes").countDocuments());
    log.info("activemq msg={}", listener.getMsgCount().get()); ②
    log.info("postgres client vote count={}", ③
        jdbcTemplate.queryForObject("select count(*) from vote", Long.class));
}
```

- ① getting the count of vote documents from MongoDB client
- ② getting number of messages received from JMS listener
- ③ getting the number of vote rows from Postgres client

419.8. Client Status Output

The following shows an example of the client output in the `@BeforeEach` method, captured after the first test and before the second test.

Example Client Status Output

```
ElectionCNTTest#init:85 mongo client vote count=6
ElectionCNTTest#init:87 activemq msg=6
ElectionCNTTest#init:88 postgres client vote count=6
```

Very complete!

Chapter 420. Summary

In this module we learned:

- how to more seamlessly integrate Docker and DockerCompose into unit tests using Testcontainers library
- how to inject dynamically assigned values into the application context to allow them to be injected into components at startup
- to execute shell commands from a JUnit test into a running container using Testcontainers library
- to establish client connection to back-end resources from our JUnit JVM operating the unit test
 - in the event that we need this information to verify test success or simply perform some debug of the scenario

Although integration tests should never 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. For example, it was not until I wrote and executed the integration tests in this lecture that I discovered I was accidentally using JMS queuing semantics versus topic semantics between the two services. When I added the extra JMS listener—the Elections Service suddenly started loosing messages. Good find!!

Testcontainers with Spock

copyright © 2021 jim stafford (jim.stafford@jhu.edu)

Chapter 421. Introduction

In several other lectures in this section I have individually covered the use of embedded resources, Docker, Docker Compose, and Testcontainers for the purpose of implementing integration tests using JUnit Jupiter.

In this lecture, I am going to cover using [Docker Compose](#) and [Testcontainers](#) with [Spock](#) to satisfy an additional audience. I am assuming the reader of this set of lecture notes may not have gone through the earlier material but is familiar with Docker and Spock (but not used together). I will be repeating some aspects of earlier lectures but provide only light detail. Please refer back to the earlier lecture notes if you need more details.

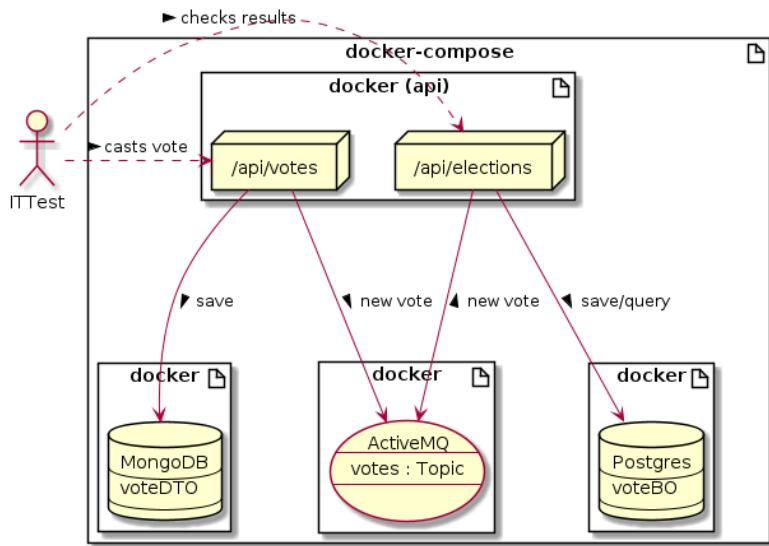


Figure 178. Target Integration Environment

Integration Unit Test terminology

I use the term "integration test" somewhat loosely but use the term "integration unit test" to specifically mean a test that uses the Spring context under the control of a simple unit test capable of being run inside of an IDE (without assistance) and executed during the [Maven test phase](#). I use the term "unit test" to mean the same thing except with stubs or mocks and the lack of the overhead (and value) of the Spring context.



421.1. Goals

You will learn:

- to identify the capability of Docker Compose to define and implement a network of virtualized services running in Docker
- to identify the capability of Testcontainers to seamlessly integrate Docker and Docker Compose into unit test frameworks including Spock
- to author end-to-end, integration unit tests using Spock, Testcontainers, Docker Compose, and Docker
- to implement inspections of running Docker images
- to implement inspects of virtualized services during tests
- to instantiate virtualized services for use in development

421.2. Objectives

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

1. define a simple network of Docker-based services within Docker Compose
2. control the lifecycle of a Docker Compose network from the command line
3. implement a Docker Compose override file
4. control the lifecycle of a Docker Compose network using Testcontainers
5. implement an integration unit test within Spock, using Testcontainers and Docker Compose
6. implement a hierarchy of test classes to promote reuse

Chapter 422. Background

422.1. Application Background

The application we are implementing and looking to test is a set of voting services with back-end resources. Users cast votes using the Votes Service and obtain election results using the Elections Service. Casted votes are stored in MongoDB and election results are stored and queried in Postgres. The two services stay in sync through a JMS topic hosted on ActiveMQ.

Because of deployment constraints unrelated to testing—the two services have been hosted in the same JVM

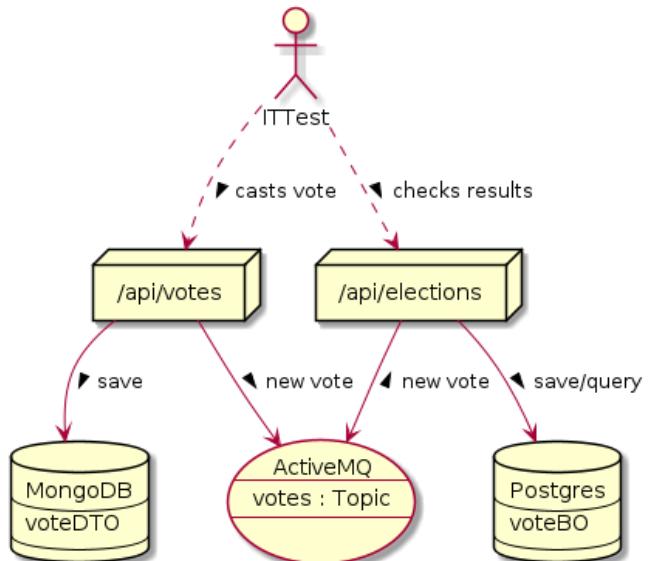


Figure 179. Voting and Election Services

422.2. Integration Testing Approach

The target of this lecture is the implementation of end-to-end integration tests. Integration tests do not replace fine-grain unit tests. In fact there are people with strong opinions ([expressed](#)) that believe any attention given to integration tests takes away from the critical role of unit tests when it comes to thorough testing. I will agree there is some truth to that—we should not get too distracted by this integration verification playground to the point that we end up placing tests that could be verified in pure, fast unit tests—inside of larger, slower integration tests. However, there has to be a point in the process where we need to verify some amount of useful end-to-end threads of our application in an **automated** manner—especially in today's world of microservices where critical supporting services have been broken out. Without the integration test—there is nothing that proves everything comes together during dynamic operation. Without the automation—there is no solid chance regression testing.

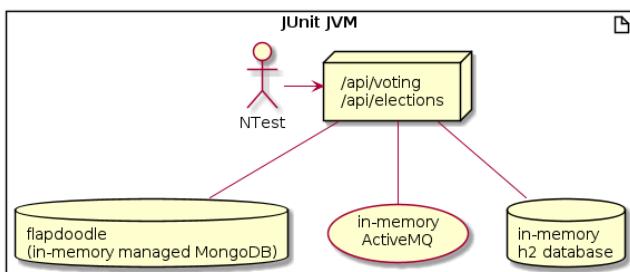


Figure 180. In-Memory/Simulated Integration Testing Environment

One way to begin addressing automated integration testing with back-end resources is through the use of in-memory configurations and simulation of dependencies—local to the unit test JVM. This addresses some of the integration need when it is something like a database or JMS server, but will miss the mark completely when we need particular versions of a full fledged application service.

We want to instead take advantage of the popularity of Docker and the ability to virtualize most back-end and many application services. We want/need this to be automated like our other tests so that they can be run as a part of any build or release. Because of their potential extended length of time and narrow focus—we will want to separate them into distinct modules to control when they are executed.

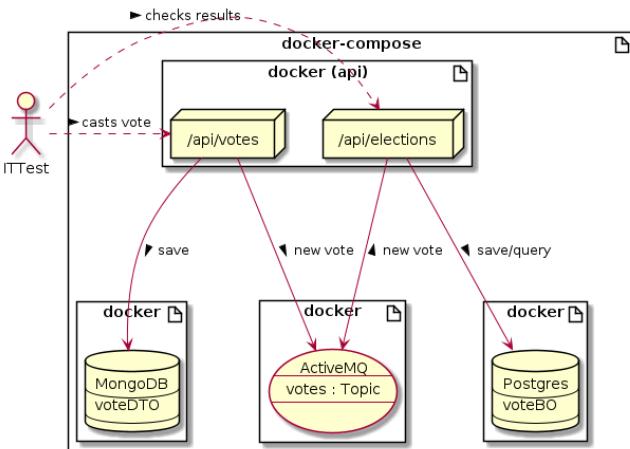


Figure 181. Virtualized Integration Testing Environment

422.3. Docker Compose

A network of services can be complex and managing many individual Docker images is clumsy. It would be best if we took advantage of a Docker network/service management layer called Docker Compose.

Docker Compose uses a YAML file to define the network, services, and even builds services with "source" build information. With that in place, we can issue a build, start and stop of the services as well as execute commands to run within the running images. All of this must be on the same machine.

Because Docker Compose is limited to a single machine and is primarily just a thin coordination layer around Docker—it is MUCH simpler to use than Kubernetes or MiniKube. For those familiar with Kubernetes—I like to refer to it as a "poor man's Helm Chart".

At a minimum, Docker Compose provides a convenient wrapper where we can place environment and runtime options for individual containers. These containers could be simple databases or JMS servers—eliminating the need to install software on the local development machine. The tool really begins to shine when we need to define dependencies and communication paths between services.

422.4. Testcontainers

Testcontainers provides a seamless integration of Docker and Docker Compose into unit test frameworks—including JUnit 4, JUnit 5, and Spock. Testcontainers manages a library of resource-specific containers that can provide access to properties that are specific to a particular type of image (e.g., databaseUrl for a Postgres container). Testcontainers also provide a generic container and a Docker Compose container—which provide all the necessary basics of running either a single image or a network of images.

Testcontainers provides features to

- parse the Docker Compose file to learn the configuration of the network

- assign optional variables used by the Docker Compose file
- expose specific container ports as random host ports
- identify the host port value of a mapped container port
- delay the start of tests while built-in and customizable "wait for" checks execute to make sure the network is up and ready for testing
- execute shell commands against the running containers
- share a running network between (possibly ordered) tests or restart a dirty network between tests

Chapter 423. Docker Compose

Before getting into testing, I will cover Docker Compose as a stand-alone capability. Docker Compose is very useful in standing up one or more Docker containers on a single machine, in a development or integration environment, without installing any software beyond Docker and Docker Compose (a simple binary).

423.1. Docker Compose File

Docker Compose uses one or more `YAML` Docker Compose files for configuration. The default primary file name is `docker-compose.yml`, but you can reference any file using the `-f` option.

The following is a Docker Compose File that defines a simple network of services. I reduced the version of the file in the example to `2` versus a current version of `3.8` since what I am demonstrating has existed for many (>5) years.

I have limited the service definitions to an image spec, environment variables, and dependencies. I have purposely not exposed any container ports at this time to avoid concurrent execution conflicts in the base file. I have also purposely left out any build information for the API image since that should have been built by an earlier module in the Maven dependencies. However, you will see a decoupled way to add port mappings and build information shortly when we get to the Docker Compose Override/Extend topic. For now — this is our core network definition.

Example docker-compose.yml File

```
version: '2'
services:
  mongo:
    image: mongo:4.4.0-bionic
    environment:
      MONGO_INITDB_ROOT_USERNAME: admin
      MONGO_INITDB_ROOT_PASSWORD: secret
  postgres:
    image: postgres:12.3-alpine
    environment:
      POSTGRES_PASSWORD: secret
  activemq:
    image: rmohr/activemq:5.15.9
  api:
    image: dockercompose-votes-api:latest
    depends_on: ①
      - mongo
      - postgres
      - activemq
    environment:
      - spring.profiles.active=integration
      - MONGODB_URI=mongodb://admin:secret@mongo:27017/votes_db?authSource=admin
      - DATABASE_URL=postgres://postgres:secret@postgres:5432/postgres
```

① defines a requirement as well as an `/etc/hostname` entry to dependent

423.2. Start Network

We can start the network using the `up` command. We can add a `-d` option to make all services run in the background. The runtime container names will have a project prefix and that value defaults to the name of the parent directory. It can be overridden using the `-p` option.

Starting Explicitly Named Network in Background

```
$ docker-compose -p foo up -d
Creating foo_activemq_1 ... done
Creating foo_postgres_1 ... done
Creating foo_mongo_1    ... done
Creating foo_api_1      ... done
```

The following shows the runtime Docker image name and port numbers for the running images. They all start with the project prefix "foo". This is important when trying to manage multiple instances of the network. Notice too that none of the ports have been mapped to a host port at this time. However, they are available on the internally defined "foo" network (i.e., accessible from the API service).

Partial Docker Status

IMAGE	PORTS	NAMES
dockercompose-votes-api:latest		foo_api_1
postgres:12.3-alpine	5432/tcp	foo_postgres_1
rmohr/activemq:5.15.9	1883/tcp, 5672/tcp, ...	foo_activemq_1
mongo:4.4.0-bionic	27017/tcp	foo_mongo_1

① no internal container ports are being mapped to localhost ports at this time

423.3. Access Logs

You can access the logs of all running services or specific services running in the background using the `logs` command and by naming the services desired. You can also limit the historical size with `--tail` option and follow the log with `-f` option.

Example Access to Logs

```
$ docker-compose -p foo logs --tail 2 -f mongo activemq
Attaching to foo_activemq_1, foo_mongo_1
mongo_1  | {"t": {"$date": "2020-08-15T14:10:20.757+00:00"}, "s": "I", ...
mongo_1  | {"t": {"$date": "2020-08-15T14:11:41.580+00:00"}, "s": "I", ...
activemq_1 | INFO | No Spring WebApplicationInitializer types detected ...
activemq_1 | INFO | jolokia-agent: Using policy access restrictor classpath:...
```

423.4. Execute Commands

You can execute commands inside a running container. The following shows an example of running the Postgres CLI (`psql`) against the `postgres` container to issue a SQL command against the `VOTE` table. This can be very useful during test debugging—where you can interactively inspect the state of the databases during a breakpoint in the automated test.

Example Exec Command

```
$ docker-compose -p foo exec postgres psql -U postgres -c "select * from VOTE"
 id | choice | date | source
-----+-----+-----+
 (0 rows)
```

① executing command that runs inside the running container

423.5. Shutdown Network

We can shutdown the network using the `down` command or `<ctl>-C` if it was launched in the foreground. The project name is required if it is different from the parent directory name.

```
$ docker-compose -p foo down
Stopping foo_api_1    ... done
Stopping foo_activemq_1 ... done
Stopping foo_mongo_1   ... done
Stopping foo_postgres_1 ... done
Removing foo_api_1    ... done
Removing foo_activemq_1 ... done
Removing foo_mongo_1   ... done
Removing foo_postgres_1 ... done
Removing network foo_default
```

423.6. Override/Extend Docker Compose File

If CLI/shell access to the VMs is not enough, we can create an override file to specialize the base file. The following example maps key ports in each Docker container to a host port.

Example Docker Compose Override File

```
version: '2'
services:
  mongo: ①
    ports:
      - "27017:27017"
  postgres:
    ports:
      - "5432:5432"
  activemq:
    ports:
      - "61616:61616"
      - "8161:8161"
  api:
    build: ②
      context: ../dockercompose-votes-svc
      dockerfile: Dockerfile
    ports:
      - "${API_PORT}:8080"
```

① extending definitions of services from base file

② adding source module info to be able to rebuild image from this module

423.7. Using Mapped Host Ports

Mapping container ports to host ports is useful if you want to simply use Docker Compose to manage a development environment or you have a tool—like MongoDB Compass—that requires a standard URL.

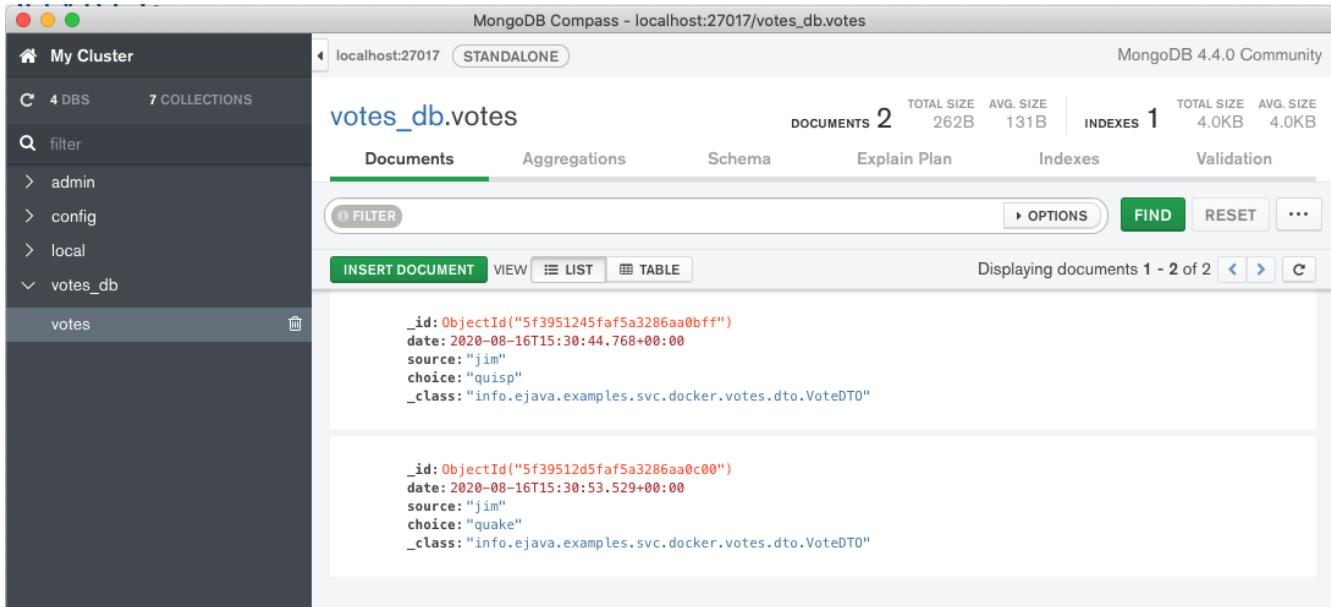


Figure 182. MongoDB Compass Connected to MongoDB in Docker Compose

423.8. Supplying Properties

Properties can be passed into the image by naming the variable. The value is derived from one of the following (in priority order):

1. `NAME: value` explicitly supplied in the Docker Compose File
2. `NAME=value` defined in environment variable
3. `NAME=value` defined in an environment file

The following are example environment files mapping `API_PORT` to either 9999 or 9090. We can activate an environment file using the `--env-file` option or have it automatically applied when named `.env`.

Example alt-env File

```
$ cat alt-env ①
API_PORT=9999
$ cat .env ②
API_PORT=9090
```

① used when `--env-file alt-env` supplied

② used by default

423.9. Specifying an Override File

You can specify an override file by specifying multiple Docker Compose files in priority order with the `-f` option. The following will use `docker-compose.yml` as a base and apply the augmentations from `development.yml`.

Example Explicit Override Specification

```
$ docker-compose -p foo -f ./docker-compose.yml -f ./development.yml up -d
Creating network "foo_default" with the default driver
```

You can have the additional file applied automatically if named `docker-compose.override.xml`. The example below uses the `docker-compose.xml` file as the primary and the `docker-compose.override.yml` file as the override.

Using Default Docker Compose and Docker Compose Override File Names

```
$ ls docker-compose*
docker-compose.override.yml docker-compose.yml
$ docker-compose -p foo up -d ①
```

① using default Docker Compose file with default override file

423.10. Override File Result

The following shows the new network configuration that shows the impact of the override file. Key communication ports of the back-end resources have been exposed on the localhost network.

Example Docker Compose Network Status with Override

\$ docker ps	① ②	
IMAGE	POR	NAMES
dockercompose-votes-api:latest	0.0.0.0:9090->8080/tcp	foo_api_1
mongo:4.4.0-bionic	0.0.0.0:27017->27017/tcp	foo_mongo_1
rmohr/activemq:5.15.9	1883/tcp, ... 0.0.0.0:61616->61616/tcp	foo_activemq_1
postgres:12.3-alpine	0.0.0.0:5432->5432/tcp	foo_postgres_1

① container ports are now mapped to (fixed) host ports

② API host port used the variable defined in `.env` file

Override files cannot reduce or eliminate collections



Override files can replace single elements but can only augment multiple elements. That means one cannot eliminate exposed ports from a base configuration file. Therefore it is best to keep from adding properties that may be needed in the base file versus adding to environment-specific files.

Chapter 424. Testcontainers and Spock

With an understanding of Docker Compose and a few Maven plugins—we could easily see how we could integrate our Docker images into an integration test using the Maven integration-test phases.

However, by using Testcontainers — we can integrate Docker Compose into our unit test framework much more seamlessly and launch tests in an ad-hoc manner right from within the IDE.

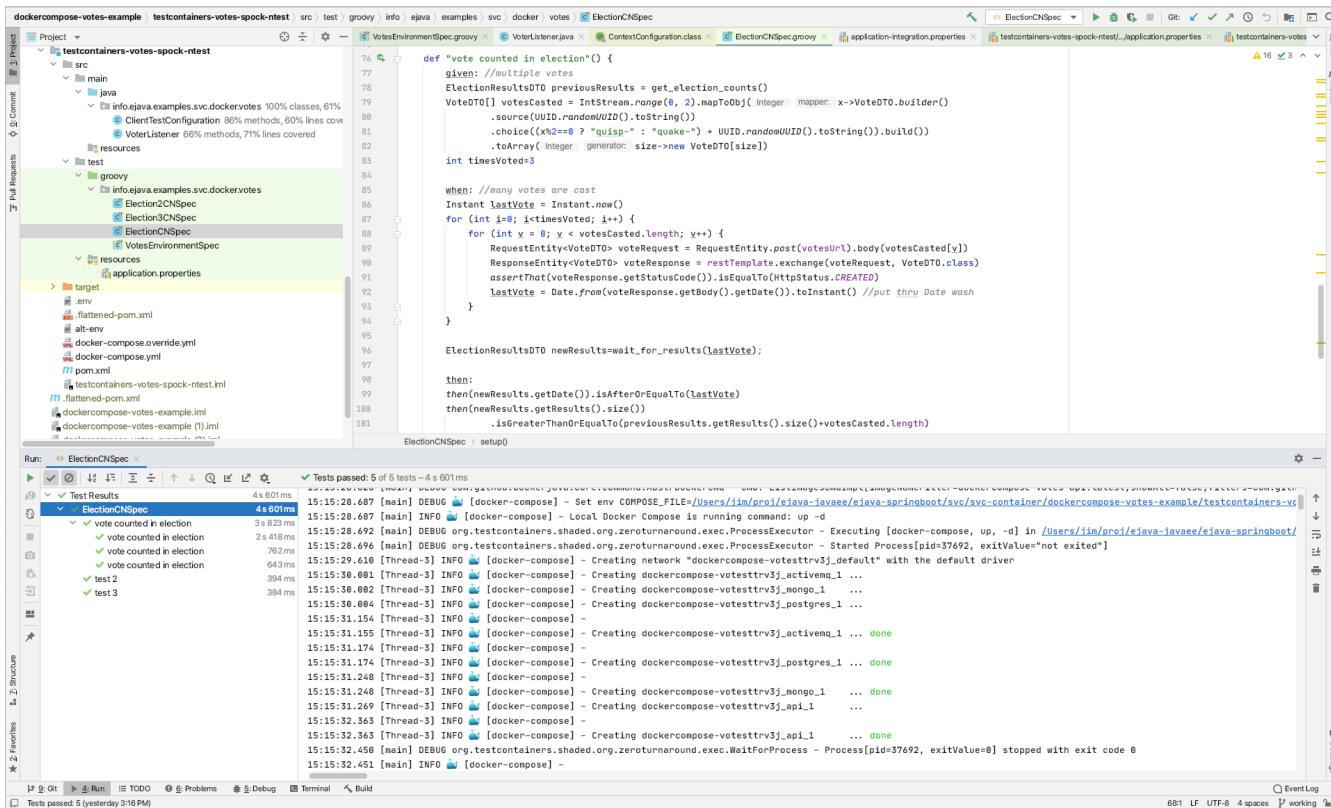


Figure 183. Example IDE Test Execution with Testcontainers and Docker Compose

424.1. Source Tree

The following shows the structure of the example integration module. We have already been working with the Docker Compose files at the root level in the previous section. Those files can be placed within the `src` directories if not being used interactively for developer commands—to keep the root less polluted.

This is an integration test-only module, so there will be no application code in the `src/main` tree. I took the opportunity to place common network helper code in the `src/main` tree to mimic what might be packaged up into test module support JAR if we need this type of setup in multiple test modules.

The `src/test` tree contains files that are specific to the specific integration tests performed. I also went a step further and factored out a base test class and then copied the initial `ElectionCNSpec` test case to demonstrate reuse within a test case and shutdown/startup in between test cases.

Example Integration Unit Test Tree

```
|-- alt-env
|-- docker-compose.override.yml
|-- docker-compose.yml
|-- pom.xml
\-- src
    |-- main
    |   |-- java
    |   |   '-- info
    ...
    |   |       '-- votes
    |   |           |-- ClientTestConfiguration.java
    |   |           '-- VoterListener.java
    |   '-- resources
    '-- test
        |-- groovy
        |   '-- info
    ...
        |       '-- votes
        |           |-- VotesEnvironmentSpec.groovy
        |           |-- ElectionCNSpec.groovy
        |           |-- Election2CNSpec.groovy
        |           '-- Election3CNSpec.groovy
        '-- resources
            '-- application.properties
```

424.2. @SpringBootConfiguration

Configuration is being supplied to the tests by the `ClientTestConfiguration` class. The following shows some traditional `@Value` property value injections that could have also been supplied through a `@ConfigurationProperties` class. We want these values set to the assigned host information at runtime.

Traditional @SpringBootConfiguration

```
@SpringBootConfiguration()
@EnableAutoConfiguration
public class ClientTestConfiguration {
    @Value("${it.server.host:localhost}")
    private String host; ①
    @Value("${it.server.port:9090}")
    private int port; ②
    ...
}
```

① value is commonly `localhost`

② value is **dynamically** generated at runtime

424.3. Traditional @Bean Factories

The configuration class supplies a traditional set of `@Bean` factories with base URLs to the two services. We want the later two URIs injected into our test. So far so good.

@Bean Factories

```
//public class ClientTestConfiguration { ...
@Bean
public URI baseUrl() {
    return UriComponentsBuilder.newInstance()
        .scheme("http").host(host).port(port).build().toUri();
}
@Bean
public URI votesUrl(URI baseUrl) {
    return UriComponentsBuilder.fromUri(baseUrl).path("api/votes")
        .build().toUri();
}
@Bean
public URI electionsUrl(URI baseUrl) {
    return UriComponentsBuilder.fromUri(baseUrl).path("api/elections")
        .build().toUri();
}
@Bean
public RestTemplate anonymousUser(RestTemplateBuilder builder) {
    RestTemplate restTemplate = builder.build();
    return restTemplate;
}
```

424.4. DockerComposeContainer

In order to obtain the assigned port information required by the URI injections, we first need to define our network container. The following shows a set of static helper methods that locates the Docker Compose file, instantiates the Docker Compose network container, assigns it a project name, and exposes container port `8080` from the API to a random available host port.

During network startup, Testcontainers will also wait for network activity on that port before returning control back to the test.

```
public static File composeFile() {
    File composeFile = new File("./docker-compose.yml"); ①
    Assertions.assertThat(composeFile.exists()).isTrue();
    return composeFile;
}

public static DockerComposeContainer testEnvironment() {
    DockerComposeContainer env =
        new DockerComposeContainer("dockercompose-votes", composeFile())
            .withExposedService("api", 8080);
    return env;
}
```

- ① Testcontainers will fail if Docker Compose file reference does not include an explicit parent directory (i.e., `./` is required)

Mapped Volumes may require additional settings

Testcontainers automatically detects whether the test is being launched from within or outside a Docker image (outside in this example). Some additional [tweaks to the Docker Compose file](#) are required only if disk volumes are being mapped. These tweaks are called forming a "wormhole" to have Docker spawn sibling containers and share resources. We are not using volumes and will not be covering the wormhole pattern here.



424.5. @SpringBootTest

The following shows an example `@SpringBootTest` declaration. The test is a pure client to the server-side and contains no service web tier. The configuration was primarily what I just showed you—being primarily based on the URIs.

The test uses an optional `@Stepwise` orchestration for tests in case there is an issue sharing the dirty service state that a known sequence can solve. This should also allow for a lengthy end-to-end scenario to be broken into ordered steps along test method boundaries.

Here is also where the URIs are being injected—but we need our network started before we can derive the ports for the URIs.

Example @SpringBootTest Declaration

```
@SpringBootTest(classes = [ClientTestConfiguration.class],  
    webEnvironment = SpringBootTest.WebEnvironment.NONE)  
@Stepwise  
@Slf4j  
@DirtiesContext  
abstract class VotesEnvironmentSpec extends Specification {  
    @Autowired  
    protected RestTemplate restTemplate  
    @Autowired  
    protected URI votesUrl  
    @Autowired  
    protected URI electionsUrl  
  
    def setup() {  
        log.info("votesUrl={}", votesUrl) ①  
        log.info("electionsUrl={}", electionsUrl)  
    }  
}
```

① URI injections—based on dynamic values—must occur before tests

424.6. Spock Network Management

Testcontainers management within Spock is more manual than with JUnit—mostly because Spock does not provide first-class framework support for static variables. No problem, we can find many ways to get this to work. The following shows the network container being placed in a `@Shared` property and started/stopped at the Spec level.

Set System Property in setupSpec()

```
@Shared ①  
protected DockerComposeContainer env = ClientTestConfiguration.testEnvironment()  
  
def setupSpec() {  
    env.start() ②  
}  
def cleanupSpec() {  
    env.stop() ③  
}
```

① network is instantiated and stored in a `@Shared` variable accessible to all tests

② test case initialization starts the network

③ test case cleanup stops the network

But what about the dynamically assigned port numbers? We have three ways that can be used to resolve them.

424.7. Set System Property

During `setupSpec`, we can set System Properties to be used when forming the Spring Context for each test.

Set System Property in setupSpec() Option

```
def setupSpec() {  
    env.start() ①  
    System.setProperty("it.server.port", ""+env.getServicePort("api", 8080));  
}
```

- ① after starting network, dynamically assigned port number obtained and set as a System Property for individual test cases

In hindsight, this looks like a very concise way to go. However, there were two other options available that might be of interest in case they solve other issues that arise elsewhere.

424.8. ApplicationContextInitializer

A more verbose and likely legacy Spring way of adding the port values is through a Spring `ApplicationContextInitializer` that can get added to the Spring application context using the `@ContextConfiguration` annotation and some static constructs within the Spock test.

The network container gets initialized—like usual—except a reference to the container gets assigned to a static variable where the running container can be inspected for dynamic values during an `initialize()` callback.

```
...
import org.springframework.context.ApplicationContextInitializer;
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.boot.test.util.TestPropertyValues;

...
@SpringBootTest(...)
@ContextConfiguration(initializers = Initializer.class) ④
...

abstract class VotesEnvironmentSpec extends Specification {
    private static DockerComposeContainer staticEnv ①
    static class Initializer ③
        implements ApplicationContextInitializer<ConfigurableApplicationContext> {
            @Override
            void initialize(ConfigurableApplicationContext ctx) {
                TestPropertyValues values = TestPropertyValues.of(
                    "it.server.port=" + staticEnv.getServicePort("api", 8080))
                values.applyTo(ctx)
            }
        }
    }

    @Shared
    protected DockerComposeContainer env = ClientTestConfiguration.testEnvironment()
    def setupSpec() {
        staticEnv = env ②
        env.start()
    }
    ...
}
```

① static variable declared to hold reference to singleton network

② `@Shared` network assigned to static variable

③ `Initializer` class defined to obtain network information from network and inject into test properties

④ `Initializer` class registered with Spring application context

424.9. DynamicPropertySource

A similar, but more concise way to leverage the callback approach is to leverage the newer Spring `@DynamicPropertySource` construct. At a high level—nothing has changed with the management of the network container. Spring simply eliminated the need to create the boilerplate class, etc. when supplying properties dynamically.

```
import org.springframework.test.context.DynamicPropertyRegistry
import org.springframework.test.context.DynamicPropertySource
...
private static DockerComposeContainer staticEnv ①
@DynamicPropertySource ③
static void properties(DynamicPropertyRegistry registry) {
    registry.add("it.server.port", ()->staticEnv.getServicePort("api", 8080));
}

@Shared
protected DockerComposeContainer env = ClientTestConfiguration.testEnvironment()
def setupSpec() {
    staticEnv = env ②
    env.start()
}
```

① static variable declared to hold reference to singleton network

② @Shared network assigned to static variable

③ @DynamicPropertySource defined on a static method to obtain network information from network and inject into test properties

424.10. Resulting Test Initialization Output

The following shows an example startup prior to executing the first test. You will see TestContainers start Docker Compose in the background and then wait close to ~12 seconds for the API port 8080 to become active.

Maven/Spock Test Startup

```
13:52:28.467 DEBUG [docker-compose] - Set env COMPOSE_FILE=
    .../dockercompose-votes-example/testcontainers-votes-spock-ntest./docker-
compose.yml
13:52:28.467 INFO [docker-compose] - Local Docker Compose is running command: up -d
13:52:28.472 DEBUG org.testcontainers.shaded.org.zeroturnaround.exec.ProcessExecutor -
    Executing [docker-compose, up, -d]
...
13:52:28.996 INFO [docker-compose] - Creating network "dockercospose-
votesdkakfi_default" with the default driver
INFO [docker-compose] - Creating dockercompose-votesdkakfi_mongo_1 ...
INFO [docker-compose] - Creating dockercompose-votesdkakfi_postgres_1 ...
INFO [docker-compose] - Creating dockercompose-votesdkakfi_activemq_1 ...
INFO [docker-compose] - Creating dockercompose-votesdkakfi_activemq_1 ... done
INFO [docker-compose] - Creating dockercompose-votesdkakfi_mongo_1 ... done
INFO [docker-compose] - Creating dockercompose-votesdkakfi_postgres_1 ... done
INFO [docker-compose] - Creating dockercompose-votesdkakfi_api_1 ... done
INFO [docker-compose] - Creating dockercompose-votesdkakfi_api_1 ... done
13:52:30.803 DEBUG org.testcontainers.shaded.org.zeroturnaround.exec.WaitForProcess -
    Process...
13:52:30.804 INFO [docker-compose] - Docker Compose has finished running
... (waiting for containers to start)
13:52:45.100 DEBUG
org.springframework.test.context.support.DependencyInjectionTestExecutionListener -
    :: Spring Boot ::      (v2.3.2.RELEASE)
...
---
```

At this point, we are ready to use normal 'restTemplate' or 'WebClient' calls to test our interface to the overall application.

```
---
13:52:48.031 VotesEnvironmentSpec votesUrl=http://localhost:32838/api/votes
13:52:48.032 VotesEnvironmentSpec electionsUrl=http://localhost:32838/api/elections
```

Chapter 425. Additional Waiting

Testcontainers will wait for the exposed port to become active. We can add additional wait tests to be sure the network is in a ready state to be tested. The following adds a check for the two URLs to return a successful response.

Example Wait For URL

```
def setup() {
    /**
     * wait for various events relative to our containers
     */
    env.waitingFor("api", Wait.forHttp(votesUrl.toString())) ①
    env.waitingFor("api", Wait.forHttp(electionsUrl.toString()))
```

① test setup holding up start of test for two API URL calls to be successful

Chapter 426. Executing Commands

If useful, we can also invoke commands within the running network containers at points in the test. The following shows a CLI command invoked against each database container that will output the current state at this point in the test.

Example Execute Commands

```
/**  
 * run sample commands directly against containers  
 */  
ContainerState mongo = (ContainerState) env.getContainerByServiceName("mongo_1")  
    .orElseThrow()  
ExecResult result = mongo.execInContainer("mongo", ①  
    "-u", "admin", "-p", "secret", "--authenticationDatabase", "admin",  
    "--eval", "db.getSiblingDB('votes_db').votes.find()");  
log.info("voter votes = {}", result.getStdout()) ②  
  
ContainerState postgres = (ContainerState) env.getContainerByServiceName("postgres_1")  
    .orElseThrow()  
result = postgres.execInContainer("psql",  
    "-U", "postgres",  
    "-c", "select * from vote");  
log.info("election votes = {}", result.getStdout())
```

① executing shell command inside running container in network

② obtaining results in stdout

426.1. Example Command Output

The following shows the output of the standard output obtained from the two containers after running the CLI query commands.

Example Command Output

```
14:32:15.075 ElectionCNSpec#setup:67 voter votes = MongoDB shell version v4.4.0
connecting to:
mongodb://127.0.0.1:27017/?authSource=admin&compressors=disabled&gssapiServiceName=mon
godb
Implicit session: session { "id" : UUID("a824b7b8-634a-426b-8d21-24c5680864f6") }
MongoDB server version: 4.4.0

{ "_id" : ObjectId("5f382a2c62cb0d4f36d96cfa"),
  "date" : ISODate("2020-08-15T18:32:12.706Z"),
  "source" : "684c586f...",
  "choice" : "quisp-82...",
  "_class" : "info.ejava.examples.svc.docker.votes.dto.VoteDTO" }
{ "_id" : ObjectId("5f382a2d62cb0d4f36d96cfb"),
  "date" : ISODate("2020-08-15T18:32:13.511Z"),
  "source" : "df3a973a...",
  "choice" : "quake-5e...",
  "_class" : "info.ejava.examples.svc.docker.votes.dto.VoteDTO" }
...
14:32:15.263 main  INFO      i.e.e.svc.docker.votes.ElectionCNSpec#setup:73 election
votes =
      id          | choice       |           date        | source
-----+-----+-----+-----+-----+
5f382a2c62cb0d4f36d96cfa | quisp-82... | 2020-08-15 18:32:12.706 | 684c586f...
5f382a2d62cb0d4f36d96cfb | quake-5e... | 2020-08-15 18:32:13.511 | df3a973a...
...
(6 rows)
```

Chapter 427. Client Connections

Although an interesting and potentially useful feature to be able to execute a random shell command against a running container under test—it can be very clumsy to interpret the output when there is another way. We can—instead—establish a resource client to any of the services we need additional state from.

The following will show adding resource client capabilities that were originally added to the API server. If necessary, we can use this low-level access to trigger specific test conditions or evaluate something performed.

427.1. Maven Dependencies

The following familiar Maven dependencies can be added to the pom.xml to add the resources necessary to establish a client connection to each of the three back-end resources.

Client Connection Maven Dependencies

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-activemq</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
</dependency>
```

427.2. Hard Coded Application Properties

We can simply add the following hard-coded resource properties to a property file since this is static information necessary to complete the connections.

Hard Coded Properties

```
#activemq
spring.jms.pub-sub-domain=true

#postgres
spring.datasource.driver-class-name=org.postgresql.Driver
spring.datasource.username=postgres
spring.datasource.password=secret
```

However, we still will need the following properties added that consist of dynamically assigned values.

Dynamic Properties Needed

```
spring.data.mongodb.uri
spring.activemq.broker-url
spring.datasource.url
```

427.3. Dynamic URL Helper Methods

The following helper methods are used to form a valid URL String once the hostname and port number are known.

Dynamic URL Helper Methods

```
public static String mongoUrl(String host, int port) {
    return String.format("mongodb://admin:secret@%s:%d/votes_db?authSource=admin",
host, port);
}
public static String jmsUrl(String host, int port) {
    return String.format("tcp://%s:%s", host, port);
}
public static String jdbcUrl(String host, int port) {
    return String.format("jdbc:postgresql://%s:%d/postgres", host, port);
```

427.4. Adding Dynamic Properties

The hostname and port number(s) can be obtained from the running network and supplied to the Spring context using one of the three techniques shown earlier ([System.setProperty](#), [ConfigurableApplicationContext](#), or [DynamicPropertyRegistry](#)). The following shows the [DynamicPropertyRegistry](#) technique.

Adding Dynamic Properties

```
public static void initProperties( ①
    DynamicPropertyRegistry registry, DockerComposeContainer env) {
    registry.add("it.server.port", ()->env.getServicePort("api", 8080));
    registry.add("spring.data.mongodb.uri", ()-> mongoUrl(
        env.getServiceHost("mongo", null),
        env.getServicePort("mongo", 27017)
    ));
    registry.add("spring.activemq.broker-url", ()->jmsUrl(
        env.getServiceHost("activemq", null),
        env.getServicePort("activemq", 61616)
    ));
    registry.add("spring.datasource.url", ()->jdbcUrl(
        env.getServiceHost("postgres", null),
        env.getServicePort("postgres", 5432)
    ));
}
```

① helper method called from `@DynamicPropertySource` callback in unit test

427.5. Adding JMS Listener

We can add a class to subscribe and listen to the `votes` topic by declaring a `@Component` with a method accepting a JMS `TextMessage` and annotated with `@JmsListener`. The following example just prints debug messages of the events and counts the number of messages received.

Example JMS Listener

```
...
import org.springframework.jms.annotation.JmsListener;
import javax.jms.TextMessage;

@Component
@Slf4j
public class VoterListener {
    @Getter
    private AtomicInteger msgCount = new AtomicInteger(0);

    @JmsListener(destination = "votes")
    public void receive(TextMessage msg) throws JMSException {
        log.info("jmsMsg={}, {}", msgCount.incrementAndGet(), msg.getText());
    }
}
```

We also need to add the JMS Listener `@Component` to the Spring application context using the `@SpringBootTest.classes` property

```
@SpringBootTest(classes = [ClientTestConfiguration.class, VoterListener.class],
```

427.6. Injecting Resource Clients

The following shows injections for the resource clients. I have already showed the details behind the `VoterLister`. That is ultimately supported by the JMS AutoConfiguration and the `spring.activemq.broker-url` property.

The `MongoClient` and `JdbcClient` are directly provided by the Mongo and JPA AutoConfiguration and the `spring.data.mongodb.uri` and `spring.datasource.url` properties.

Inject Resource Clients

```
@Autowired  
protected MongoClient mongoClient  
@Autowired  
protected VoterListener listener  
@Autowired  
protected JdbcTemplate jdbcTemplate
```

427.7. Resource Client Calls

The following shows an example set of calls that simply obtains document/message/row counts. However, with that capability demonstrated—much more is easily possible.

Example Resource Client Calls

```
/**  
 * connect directly to exploded port# of images to obtain sample status  
 */  
log.info("mongo client vote count={}",  
    mongoClient.getDatabase("votes_db").getCollection("votes").countDocuments())  
log.info("activemq msg={}, listener.getMsgCount().get()")  
log.info("postgres client vote count{}",  
    jdbcTemplate.queryForObject("select count (*) from vote", Long.class))
```

The following shows the output from the example resource client calls

Example Resource Call Output

```
ElectionCNSpec#setup:54 mongo client vote count=18  
ElectionCNSpec#setup:55 activemq msg=18  
ElectionCNSpec#setup:57 postgres client vote count=18
```

Chapter 428. Test Hierarchy

Much of what I have covered can easily go into a helper class or test base class and potentially be part of a test dependency library if the amount of integration testing significantly increases and must be broken out.

428.1. Network Helper Class

The following summarizes the helper class that can encapsulate the integration between Testcontainers and Docker Compose. This class is not specific to running in any one test framework.

ClientTestConfiguration Helper Class

```
public class ClientTestConfiguration { ①
    public static File composeFile() { ...
    public static DockerComposeContainer testEnvironment() { ...
    public static void initProperties(DynamicPropertyRegistry registry,
DockerComposeContainer env) { ...
    public static void initProperties(DockerComposeContainer env) { ...
    public static void initProperties(ConfigurableApplicationContext ctx,
DockerComposeContainer env) { ...
    public static String mongoUrl(String host, int port) { ...
    public static String jmsUrl(String host, int port) { ...
    public static String jdbcUrl(String host, int port) { ...
```

① Helper class can encapsulate details of network without ties to actual test framework

428.2. Integration Spec Base Class

The following summarizes the base class that encapsulates starting/stopping the network and any helper methods used by tests. This class is specific to operating tests within Spock.

VotesEnvironmentSpec Test Base Class

```
abstract class VotesEnvironmentSpec extends Specification { ①
    def setupSpec() {
        configureEnv(env)
        ...
        void configureEnv(DockerComposeContainer env) {} ②
    def cleanupSpec() { ...
    def setup() { ...
    public ElectionResultsDTO wait_for_results(Instant resultTime) { ...
    public ElectionResultsDTO get_election_counts() { ...
```

① test base class integrates helper methods in with test framework

② extra environment setup call added to allow subclass to configure network before started

428.3. Specialized Integration Test Classes

The specific test cases can inherit all the setup and focus on their individual tests. Note that the example I provided uses the same running network within a test case class (i.e., all test methods in a test class share the same network state). Separate test cases use fresh network state (i.e., the network is shutdown, removed, and restarted between test classes).

Example Test Case

```
class ElectionCNSpec extends VotesEnvironmentSpec { ①
    @Override
    def void configureEnv(DockerComposeContainer dc) { ...
    def cleanup() { ...
    def setup() { ...
    def "vote counted in election"() { ...
    def "test 2"() { ...
    def "test 3"() { ...
```

- ① concrete test cases provide specific tests and extra configuration, setup, and cleanup specific to the tests

Example Test Case 2

```
class Election2CNSpec extends VotesEnvironmentSpec {
    def "vote counted in election"() { ...
    def "test 2"() { ...
    def "test 3"() { ...
```

428.4. Test Execution Results

The following image shows the completion results of the integration tests. One thing to note with Spock is that it only seems to attribute time to a test setup/execution/cleanup and not to the test case's setup and cleanup. Active MQ is very slow to shutdown and there is easily 10-20 seconds in between test cases that is not depicted in the timing results.

Run: info.ejava.examples.svc.docker.votes in testcontainers-votes-s	
	✓
	✗
	↓
	↓
	⊸
	⊸
	⟳
	🔗
	⚙️
▼	✓ votes (info.ejava.examples.svc.docker) 11 s 343 ms
▼	✓ Election2CNSpec 3 s 858 ms
▼	✓ vote counted in election 3 s 146 ms
✓ vote counted in election 2 s 118 ms	
✓ vote counted in election 537 ms	
✓ vote counted in election 491 ms	
✓ test 2 355 ms	
✓ test 3 357 ms	
▼	✓ Election3CNSpec 4 s 390 ms
▼	✓ vote counted in election 3 s 121 ms
✓ vote counted in election 1 s 844 ms	
✓ vote counted in election 677 ms	
✓ vote counted in election 600 ms	
✓ test 2 730 ms	
✓ test 3 539 ms	
▼	✓ ElectionCNSpec 3 s 95 ms
▼	✓ vote counted in election 2 s 387 ms
✓ vote counted in election 1 s 362 ms	
✓ vote counted in election 472 ms	
✓ vote counted in election 553 ms	
✓ test 2 365 ms	
✓ test 3 343 ms	

Figure 184. Test Execution Results

Chapter 429. Summary

This lecture covered a summary of capability for Docker Compose and Testcontainers integrated into Spock to implement integrated unit tests. The net result is a seamless test environment that can verify that a network of components—further tested in unit tests—integrate together to successfully satisfy one or more end-to-end scenarios. For example, it was not until integration testing that I realized my JMS communications was using a queue versus a topic.

In this module we learned:

- to identify the capability of Docker Compose to define and implement a network of virtualized services running in Docker
- to identify the capability of Testcontainers to seamlessly integrate Docker and Docker Compose into unit test frameworks including Spock
- to author end-to-end, unit integration tests using Spock, Testcontainers, Docker Compose, and Docker
- to implement inspections of running Docker images
- to implement inspects of virtualized services during tests
- to instantiate virtualized services for use in development
- to implement a hierarchy of test classes to promote reuse