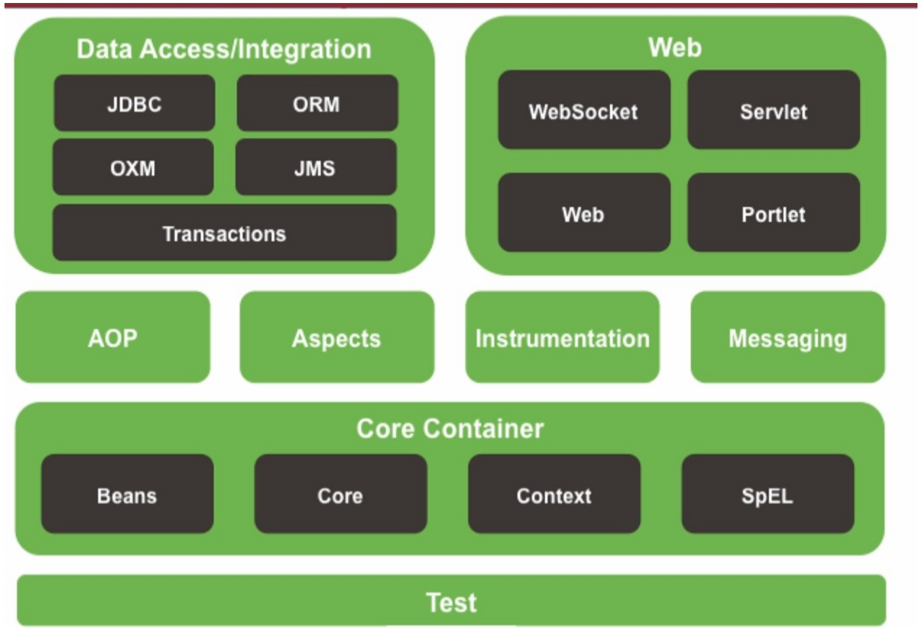
Spring 6.0 From Beginner to Guru

Introduction to Spring

- Components
  - Spring Boot -- Auto-configuration – If a specific library is on the classpath
  - Projects
    - Spring Data – Collection of project for persisting data to SQL and NoSQL databases
    - Spring Cloud
    - Spring Security – Authentication and Authorization
    - Spring Integration – Enterprise Integration Patterns
    - Spring Batch – Batch Processing
    - Spring State Machine – Open Source State Machine
  -



Spring Initializer

- For the project
  - Project name spring-6-webapp
  - Use jar (fat jar) and Java 17
  - Dependencies
    - Lombok
    - Spring Web
    - Spring Data JPA
    - H2 Database (in memory database)

JPA Relationships Equity In Hibernate

- Mapped by – The property name of other used in the relationship
- @Join Table will join the table based on specific @Join Columns
- Equity in Hibernate
  - Set Equal/Hashcode method since Hibernate use it to determine object quity
  - Strategies either id or the entire class

Initialization Data with Spring
Introduction to H2 Database Console
Introduciton of Spring MVC

- CommandLineRunner – provided by Spring Boot to execute the run method
  - Every time Spring Starts up
- Introduction to H2 Database Console
  - If you included Spring Developer Tools it is enabled by default
  - Find the URI that starts with jdbc:h2 and select that string
  - Url: http://localhost:8080/h2Console
  - Put the String into the JDBC URL
- Introduction to Spring MVC
  - Definitions
    - Model – Simple POJO passed between view and the business logic
    - View – Data requested by the client.
  -

Thymeleaf Templates

- Introduction to Thymeleaf
  - https://medium.com/thymeleaf-basics-concepts/thymeleaf-1ef952db0740

Solid Principal of OOP

- 5 Principles
  - S Single Responsibility Principal
    - Every class should have a single responsibility
      - There should never be more than one reason for a class to change
    - Classes should be small
    - Avoid God Classes
    - Split big classes into smaller classes
  - O Open Closed Principal
    - Your classes should be open for extension, but closed for modification
    - Be able to extend a classes behavior without modifying it
    - Use private variables with getters and setters – only when you need them
    - Use Abstract Base Classes
  - L Liskov Substitution Principal
    - Objects in a program would be replaceable with instances of their subtypes without altering the correctness of the prgram
    - Violations will often fail the “Is A “ Test : A square is a rectangle , but a rectangle is not a square
  - I Interface Segregation Principle
  - D Dependency Inversion Principal

Solid Principal of OOP

- I Interface Segregation Principle
  - Make sure fined grained interface that are not client specific
  - Keep your components focused and minimize dependencies between them
  - Relationship to the Singe Responsibility Principle
  - Avoid God Interfaces
- D Dependency Inversion Principal
  - Abstractions should not depend on the details and details should not depend the abstraction
    - Should work together and not be tightly coupled
  - Higher and Lower level object depended on the abstract interaction
  - Not the same as dependency injection

The Spring Context
Spring Test Context

- Default Behavior for Spring Boot
  - Configures Spring for annotated (ex. @Compoent) components I the package it is in and below
- Spring Text Context
-

Basics of Dependency Injection

- Basics of Dependency Injection
  - Dependency Injection is where a needed dependency is injected by another object
  - Can be instantiated via constructor or after via setter
  - Types of Dependency Injection
    - By class properties – Don’t use very hard to test
    - By Setters
    - By Constructor
  - Object cannot exist without that dependency being passed into it
  - Concrete classes vs interfaces
    - Interface highly preferred
    - Allows runtime to decide implementation to inject
    - Follows interface segregation principle of SOLID
    - Allows cost to be more testable and Mocks become more trivial

Basics of Dependency Injection

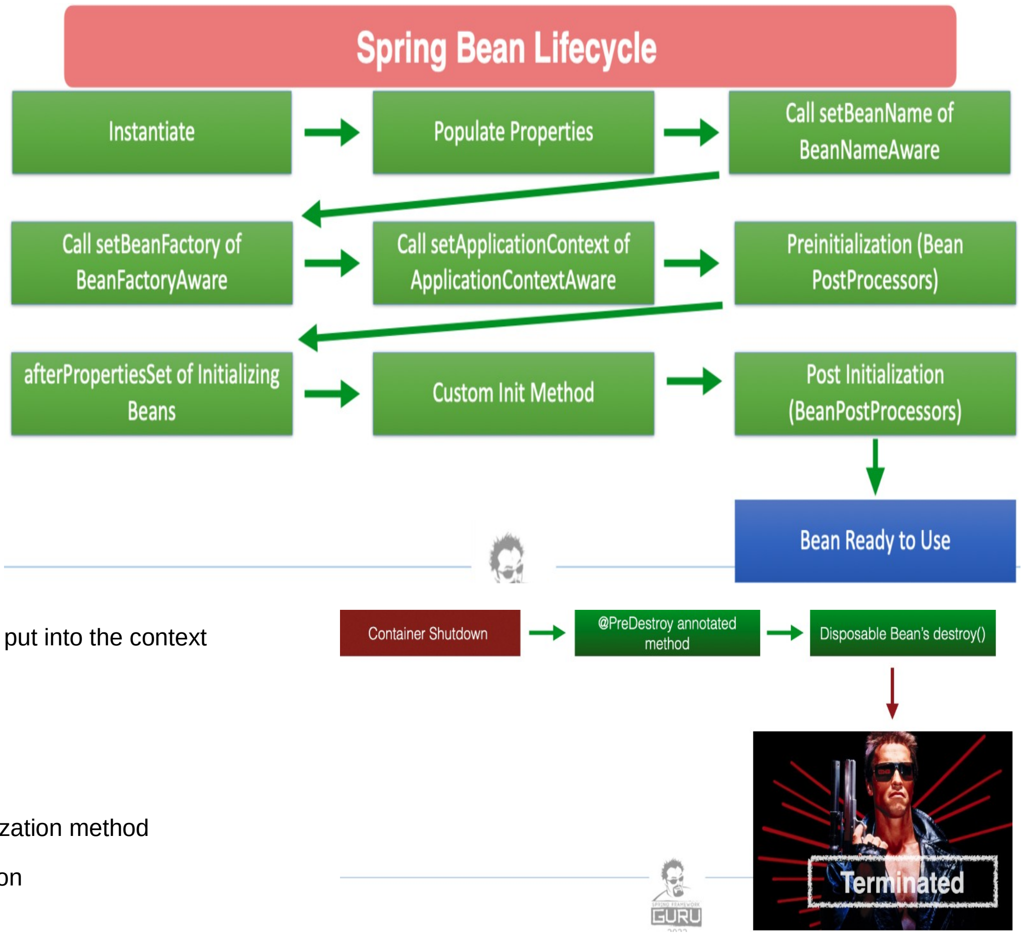
- Inversion of Control
  - A technique that allows dependencies to be injected at runtime
  - Dependencies are not predetermined since it allows the framework to |compose the application by controlling which implementation in injected
    - H2 in memory data source or MySQL data source
- Inversion of Control vs Dependency Injection
  - Dependency Injection refers to the composition of your classes
  - IoC is the runtime environment of you code
    - Control of Dependency Injection is inverted to the framework
    - Spring in control of the injection of dependencies
- Best Practices
  - Final properties for injected components
    - Declare private final and initialize in the constructor
  - Whenever practical code to an interface
  -
  -

Dependency using Spring Framework
Primary Beans
Using Qualifiers
Spring Profiles

- When doing setter injection make sure the setter has the @autowired
- Primary Beans
- Using Qualifiers
- Spring Profiles
  - If profile is defined using @ActiveProfile then a exception will be generated.
  - For default profile the second parameter of @Profile second parameter must be default00

Spring Bean Life Cycle

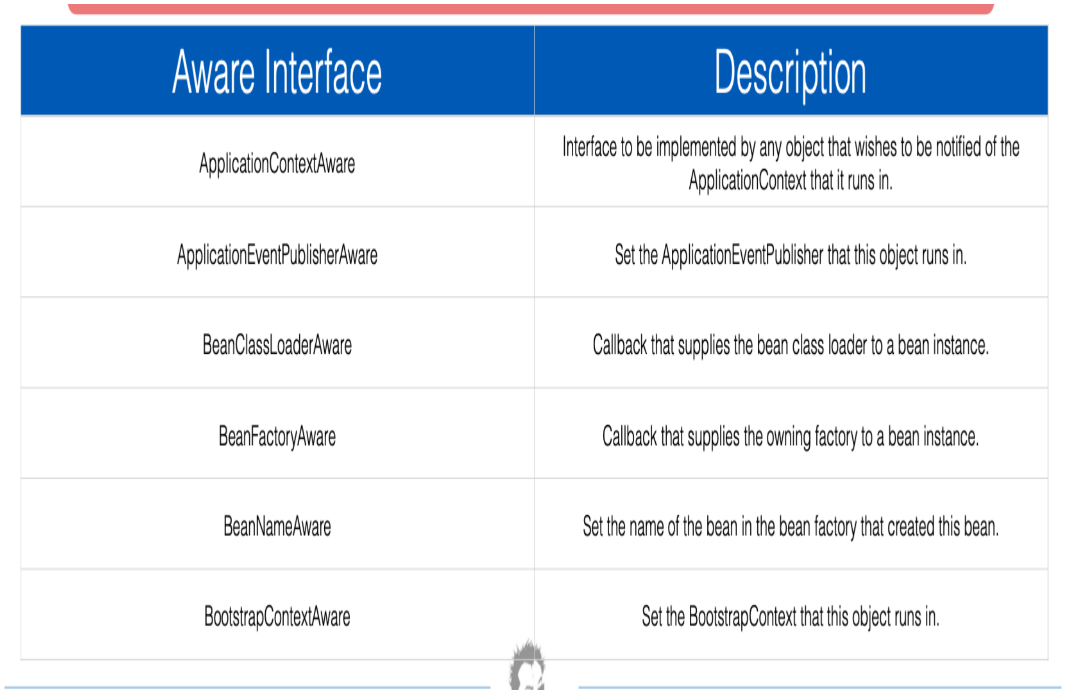
- Bean Processors are callbacks that we can hook into.
- very rare that you need to use
- Has Two interfaces you can implement for call back events
  - intializingBean.afterpropertiesSet
    - @PostConstruct
    - Called after the properties are set, but bean has not been returned to the object
  - DisposableBean.destroy()
    - @PreDestroy
    - Called before bean is destroyed by the container
- Beans Post Processors – Tap into the Spring Lifecycle and interact with beans as they are processed
  - Example Use - Create a 3rd Party Object that needs to be created, put into the context and updated (non spring managed component
  - Called for all beans in context
  - implement BeansPostProcessor
    - postProcessBeforeInitialization – Called before bean initialization method
    - postProcessAfterInitialization – Called after bean initialization



Spring Bean Lifecycle
Instantiate
Populate Properties
Call setBeanName of BeanNameAware
Call setBeanFactory of BeanFactoryAware
Call setApplicationContext of ApplicationContextAware
Preinitialization (Bean PostProcessors)
afterPropertiesSet of Initializing Beans
Custom Init Method
Post Initialization (BeanPostProcessors)
Bean Ready to Use
Container Shutdown
@PreDestroy annotated method
Disposable Bean's destroy()
Terminated
GURU

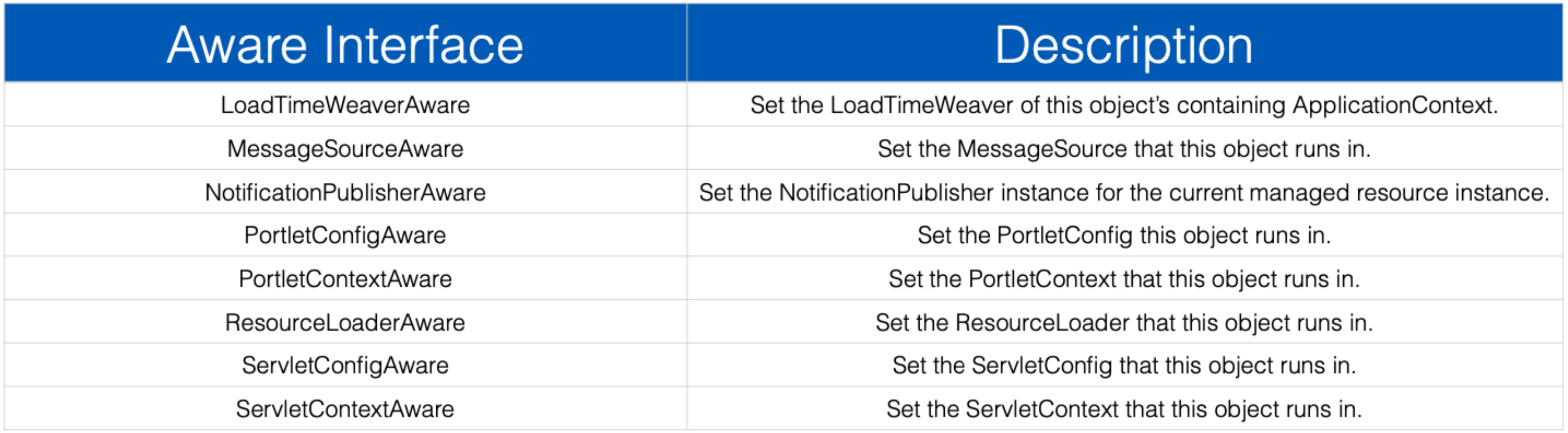
Spring Bean Life Cycle

- Aware interfaces
  - Used to access the Spring Framework Infrastructure
  - largely used in the framework
  - Rarely used by Spring Developers
  - Review extensions of the aware interfaces for current interfaces
-



Aware Interface	Description
ApplicationContextAware	Interface to be implemented by any object that wishes to be notified of the ApplicationContext that it runs in.
ApplicationEventPublisherAware	Set the ApplicationEventPublisher that this object runs in.
BeanClassLoaderAware	Callback that supplies the bean class loader to a bean instance.
BeanFactoryAware	Callback that supplies the owning factory to a bean instance.
BeanNameAware	Set the name of the bean in the bean factory that created this bean.
BootstrapContextAware	Set the BootstrapContext that this object runs in.

Spring Bean Life Cycle



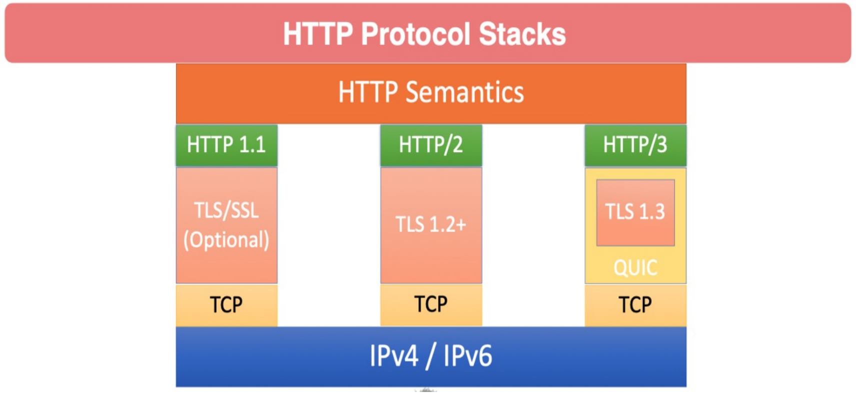
Aware Interface	Description
LoadTimeWeaverAware	Set the LoadTimeWeaver of this object's containing ApplicationContext.
MessageSourceAware	Set the MessageSource that this object runs in.
NotificationPublisherAware	Set the NotificationPublisher instance for the current managed resource instance.
PortletConfigAware	Set the PortletConfig this object runs in.
PortletContextAware	Set the PortletContext that this object runs in.
ResourceLoaderAware	Set the ResourceLoader that this object runs in.
ServletConfigAware	Set the ServletConfig that this object runs in.
ServletContextAware	Set the ServletContext that this object runs in.

HTTP Protocol

- Started as a telnet friendly protocol
  - Use ssh instead of telnet
- HTTP History : Version 1.0 from 1991 to 1995
  - Contain request line with HTTP Version number followed by headers such as User Agent
  - Response status followed by header such as Content-type, Content-length, Expires etc..
  - Http Standard were developed by IETF and W3C designed a design document
- HTTP/1.1. release in 1997 and updated in 1999/2014
  - Added support for keep alive connection, chunked encoding transfers, byte range request, transfer encoding and request pipelining
  - Added later : Request Added encoding, charset, cookies and response added encoding, charset and cookies
- HTTP 2.0 Standardized in 2015 (Performance release)
  - Transport performance was a focus
  - Improved load speed by lower latency and higher throughput
  - Helps in scaling applications.
  - High level of compatibility with HTTP/1.1 (mainly low level changes that developers won’t recognizine)

HTTP Protocol

- HTTP/3
  - Builds on concepts of HTTP/2
  - Most significant is use of the QUIC network protocol rather than TCP
- For the developers the calls, the status codes and methods are all the same.
-



HTTP Request Methods

- Request Methods (verbs) indicate the desired action to be performed.
- Get
  - Request for a resource (html file, javascript, file image)
  - Used when you visit a website
- Head
  - Ask for meta information without the body
- Post
  - Used to post data to the server (insert new data)
    - An example would be a checkout form
  - Post is a create request
- Put
  - Is a request for the enclosed entity be stored at the supplied URI. If the entity exist it is expected to be updated
  - Put is a create or update request
-

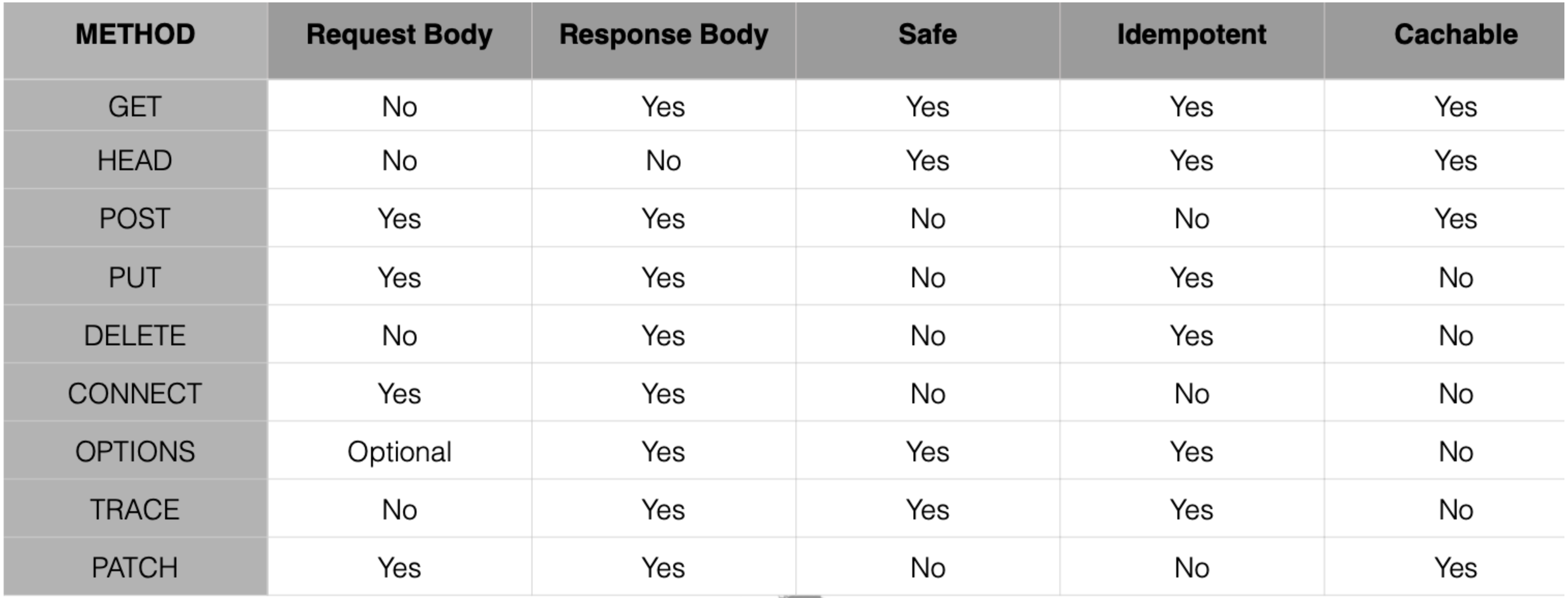
HTTP Request Methods

- Delete
  - A request to delete the specified resource
  - Not supported by the HTML Standard.
- Trace
  - Will echo the received request
  - Can be used to see if the request was altered by intermediate server
    - Example : Proxy Servers can change the request along the way
- Options
  - Returns the HTTP Methods support by the server for the specified URL
- Connect
  - Converts the request to a transparent TCP/IP tunnel typically fro HTTPS through an unencrypted HTTP Proxy
  - The author has never seen a direct use for
- Patch
  - Applies partial modifications to the specified resource

HTTP PROTOCOL

- Safe Methods
  - Considered safe to use because they only fetch information
  - Do not cause changes to the server unless you are returning dynamic data
  - Get, Head Options and Trace are all Safe Methods
- Idempotent Methods
  - A quality of an action such that repetitions of the action have no further effect on the outcome
  - put and Delete are Idempotent Methods
  - Safe Methods (Get, Head, Trace, Options) are Idempotent
  - Being truly idempotent is not enforced by the protocol, but part of the standard
- Non-Idempotent Methods
  - Post is not Idempotent
  - Multiple post are likely to create multiple resource
    - Example. Ever seen websites asking you to click submit only once
- Cacheable : Get, Head, Post, Patch
- Non Cacheable : Put, Delete, Connect Options, Trace

HTTP Protocol



METHOD	Request Body	Response Body	Safe	Idempotent	Cachable
GET	No	Yes	Yes	Yes	Yes
HEAD	No	No	Yes	Yes	Yes
POST	Yes	Yes	No	No	Yes
PUT	Yes	Yes	No	Yes	No
DELETE	No	Yes	No	Yes	No
CONNECT	Yes	Yes	No	No	No
OPTIONS	Optional	Yes	Yes	Yes	No
TRACE	No	Yes	Yes	Yes	No
PATCH	Yes	Yes	No	No	Yes

HTTP Protocol

- Status Codes
  - 100 Informational
  - 200 Indicates successful request
  - 300 series are redirections Example a website has moved
    - They are rarely used in Rest Services
  - 400 client errors
  - 500 server side errors
- Common HTTP Status Codes

200	Okay	
201	Created	
204	Accepted	
301	Move Permanently	Will get the new location back,
400	Bad Request	Example a media type that the media does not accept – Bad Data Received
401	Not Authorized	User Authentication, not authorized
404	Not Found	
405	Method Not Allowed	Method has not been found for the HTTP Verb and Context Path
500	Internal Server Error	
503	Service Unavailable	

Restful Web Services

- REST – Representational State Transfer
  - Representation – Typically JSON or XML
  - State Transfer – Typically HTTP
- Rest APIs use HTTP verbs to create, manage and delete server resources
  - Data Structures represented by JSON or XML
  - HTTP Status Code are used to communicate success, failure or errors
  - Not a standard unlike SOAP
  - kubectl get all --selector app=myweb
- Restful Terminology
  - Verbs : HTTP Methods manages the payload of the action (JSON/XML)
  - URI – Uniform Resource Identifier – A unique string identifying a resource – http://www.example.com
  - URL – Uniform Resource Locator – A URI with network information – http://www.example.com/books/1
  - Idempotent
  - States – Does not maintain any client state
  - HATEOAS – Engine of the application state
    - Rest Client should use server provided links dynamically to discover all the available action and resources it needs.
    - The server responds with text that includes hyperlinks to other actions that are currently available

Richardson Maturity Model

- Used to describe the quality of the Restful Service
- Levels
  - Level 0 : The swap of pox – Examples RPC, SOAP, XML-RPC
    - Plain old XML
    - Use Implementing protocol as a transport protocol
    - Uses one URI and one method
    - Breaks large service into distinct URIs
  - Level 1 : Resources
    - Use Multiple URIs to identity specific resources – Example http://www.example.com/product/1234 and http://www.example.com/product/4567
    - Still uses a single method get
  - Level 2 : HTTP Verbs (used with URIs for desired actions)
    - Examples : Get /products/1234 to return data for product 1234
    - Examples : Put /product/1234(with XML Body) to update data for product 1234
    - Most common in practical use
    - Introduces Verbs to implement actions
  - Level 3: Hypermedia Controls – Representation now contains URIs which may be useful to consumers
    - Help to explore the resource (Spring provides an implementation of HATEOS)
    - No clear standard at this time.
    - provides discoverability, making API more self documenting

Spring Framework and Restful Services

- Web Frameworks
  - Spring MVC
    - Has robust support for traditional Web Applications
    - Based on traditional Java Servlet API – Blocking and non reactive
    - Legacy
  - Spring Webflux
    - Uses project Reactor to provide reactive web services
    - Do not use Java Servlet API and is non blocking
    - Follows very close to the configuration model of Spring MVC – Easy Transition
  - WebFlux.fn – A functional programming model used to define endpoint
    - Alternate to annotation based configuration
    - Designed to rapidly and simple define microservices endpoints
  -

Spring Framework and Restful Services

- Web Client
  - Spring Rest Template
    - RestTemplate is Spring’s library for consuming RESTFUL web services
    - Highly configurable
    - Getting ready to be deprecated
  - Spring Web Client
    - Reactive Web Client
    - uses Reactor and Netty – A non blocking HTTP Client Library
  - Should be fluent in both
- Marshaling and Unmarshaling
  - Convert Java POJO to JSON or XML – Marshalling
  - Convert JSON or XML to Java Objects called Unmarshalling
  - Spring boot configures Jackson to facilitate Marshaling and Unmarshaling, but support other libraries

Spring Framework and Restful Services

- SPA – Single Page Applications
  - Frameworks : Vue, ReactJS, AngularJS, EmberJS
  - The frameworks are decoupled from Spring via the HTTP/JSON or XML Layer (for the Front End and Back End)
  - All of the these frameworks can consume RESTful APIS
  - Server side can be Spring Boot, .NET, Ruby on Rails – The restful API abstracts the implementation

Project Lombok Features

- How Lombok works
  - Hooks in via the Annotation processor API
    - The AST (raw source code) is passed to Lombok for code generation before the java compile continues
    - Thus, produces property compiled java code in conjunction with the Java Compile
  - target/classes’ you can view the compile class files
  - Intellij will decompile to show you the source code
  - If you need a custom setter just write the setter and Lomok will not create that setter
- Project Lombok and IDE’s
  - Since compile code is change and source files are not IDE’s can get confused by this
- Features

val	local variables declared final
var	mutable local variables
@NonNull	Null Check will throw NPE if parameter is null

Project Lombok Features

- @cleanup Will close() on resource in finally block
- @Getter Creates getter methods for all properties
- @Setter Create setter for all non-final properties
- @ToString
  - Generate String of classname and each filed separated by commas
  - Optional parameter to include field names
  - Optional parameter to include call to the super toString method
- @EqualsAndHashCode
  - Generates implements of equals (Object Other) and hashCode()
  - By default will use all no-static non-transient properties
  - Can optionally exclude specific properties
- @NoArgsConstructor
  - Generates no args constructor
  - Will cause compiler error if there are final fields
  - Can optionally force which will initialize final fields with 0 / false /null

Project Lombok Features

- @RequiredArgsContructor
  - Generates a constructor for all fields that are final or marked with @NonNull
  - Constructor will throw a NullPointerException if any @NonNull fields are null
- @All Args Constructor
  - Generates a constructor for all properties of class
  - Any @NonNull properties will have null checks
- @Data
  - Generates typical boilerplate for POJOs
  - Combines, @Getter, @Setter, @ToString, @EqualsAndHashCode, @RequiredArgsConstructor
  - No constructor is generated constructors have been declared
- @Value
  - The immutable variant of @Data
  - All fields are made private and final by default
- @NonNull
  - Set on parameter of method or constructor and a NullPointerException will be thrown if parameter is null

Project Lombok Features

- @Builder
  - Implements the builder patter for object creation
  - ex. Person.builder(“Adam Savage”).city(“San Francisco”).job(“MythBusters”).job(“UnchainedReaction”).build()
- @SneakyThrows
  - Throw Checked exceptions without declaring in calling method’s throw clause
- @Syncronized
  - A safer implementation of Java’s synchronized
- @Getters(lazy = true) for expensive getters
  - Will calculate value first time and cache
  - Additional gets will read from cache
- @Log – Crate a Java util logger
- @Slf4j – Creates a generic logging facade
  - Spring Boot’s default logger is LogBack
  - SLF4j is a generic logging facade

HTTP Client
Spring Developer Tools
Set Header on HTTP Response

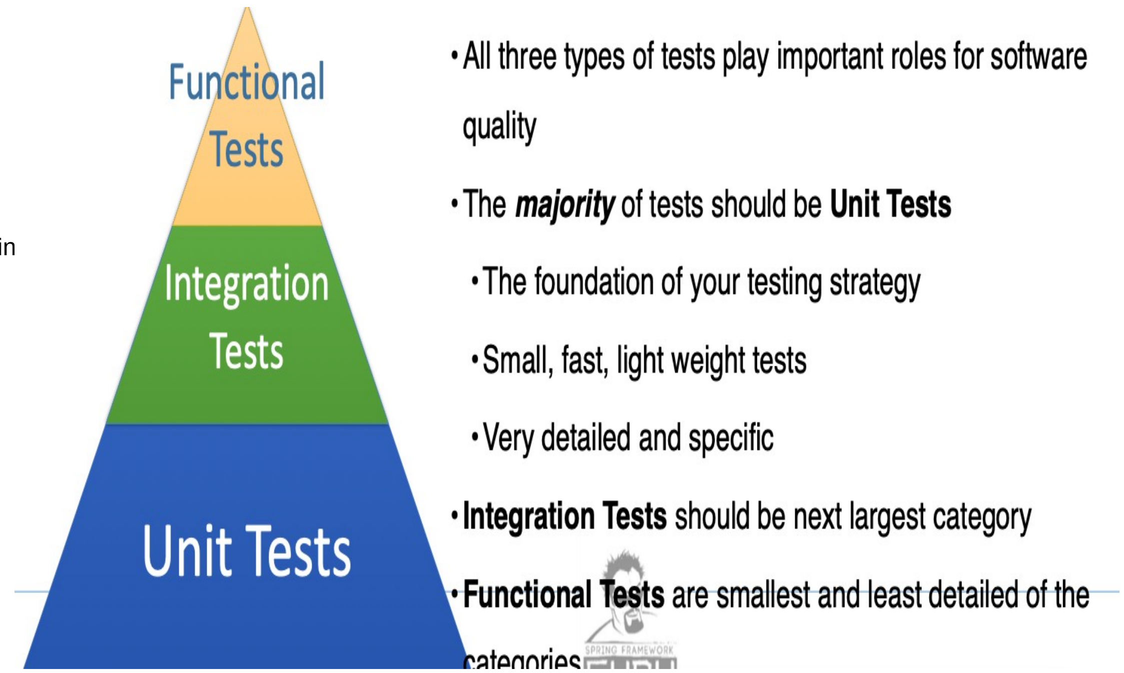
- Found in tools→ Intellij
- Spring Developer Tools
  - Live Reload for Web Applications
  - Restart Springboot when the class files change
    - For the ultimate Edition
      - Edit Configuration – Run Options – Update Classes and Resources
    - Good for small changes and cheats
- HTTP Post
  - ResponseEntity -- Extension of HttpEntity that adds an HttpStatusCode status code
    - HTTPEntity -- Represents an HTTP request or response entity, consisting of headers and body.
    - HTTPStatusCode --
- Set Header on HTTP Response
  - Use : When creating the object return the new location in the HTTP Response Header Location Attribute.
- Note on Put and Post – For Post the whole object is needed. For the Put not all fields need to be named. -- Check this one out.

Introduction to testing with MVC

- Unit Tests
  - ideal coverage is 70 to 80 percent
  - Should be small and execute very fast and light weight
  - Should have no external dependencies
    - Ex. no database, no Spring Context
- Integration test
  - Designed to test behaviors between objects and parts of the overall system
  - Much large in scope
    - Can include the Spring Context, message brokers
  - will run much slower than unit test
- Functional Tests
  - Application is live, likely deployed in a known environment
  - Functional touch points are tested
    - Using a web driver, calling web services, sending and receiving messages

Introduction to testing with MVC

- Testing Hierarchy
- Testing Spring MVC Controllers is Tricky
  - Controllers have a high degree of integration with the Spring MVC Framework
    - Request path and HTTP Method decides which method to in
    - Path Variable are parsed from path
    - JSON is bound to POJOs
    - Response is express as HTTP Response
  - Junit test are not sufficient to test the framework interaction
- Spring Mock MVC is a testing environment for testing Spring MVC Controllers
  - Provides mocks of the servlet environment
  - HTTP Request / Response, Dispatcher Servlet
  - Simulates the execution of controller as if it was running under Spring with Tomcat
- Can be run with or without the Spring Context
  - True Unit test when run without Spring Context
  - Technically an integration Test when used in conjunction with Spring Context



- All three types of tests play important roles for software quality
- The ***majority*** of tests should be **Unit Tests**
  - The foundation of your testing strategy
  - Small, fast, light weight tests
  - Very detailed and specific
- **Integration Tests** should be next largest category
- **Functional Tests** are smallest and least detailed of the categories

Introduction to testing with MVC

- Spring Boot Test Splices
  - Test Splices bring up a targeted Auto Configured Spring Boot Environment
    - Ex. Just the Database Components or just web components
    - User defined Spring beans typically not initialized
  - @WebMvcTest – A Spring Boot test splice which create a MockMVC environment for the controller or controllers under test
    - Dependencies of controllers are not included – Example User defined Spring Beans
- Using Mocks
  - Controller Dependencies must be added to the Spring Context in the test environment
  - Dependencies can be any proper implementation
    - Example of why we code to an interface
  - For testing it is common to use mock objects
  - Mocks allow you to supply a specific response for a given input
    - I.e : when method abcd is called return foo
- Mockito – A popular mocking framework for testing java
  - Mocks (Test Doubles) are alternate implementations of objects to replace real objects in tests
  - Works well for Dependency injection (mocks can be injected)
- Spring MockMvc – Allows you to test the controller interactions in a servlet context without the application running in an application server

Introduction to testing with MVC

- Types of mocks
  - Dummy – Objects are used just to get the code to compile
  - Fake – An object that has an implementation, but not production ready
  - Stub – An object with pre-defined answers to method calls
  - Mock – An object with pre-defined answers to method calls and has expectations of executions
    - Can throw an exception if an unexpected invocation is detected
    - like a stub
    - Analytics : Was it called, How many times was it called
  - Spy – In Mockito spies are mock like wrappers around the actual object
- Verify – Used to verify number of times a mocked method can be called
- Argument Matcher – Matches arguments passed to Mocked Method and will allow or disallow
- Argument Captor – Capture arguments passed to a mocked method
  - Allows performing assertions of what was passed in a method

Introduction to testing with MVC

- Testing Controllers with Mocks
  - Argument captors can be used to verify request data is properly being parsed and passed to the service layer
  - Verify interactions can be used Mocked object was called
  - Mock Return values supply data back to the controller
    - ex. object returned when getById is called on service
  - Mocks can be instructed to throw exceptions to test exception handling
-

Mock MVC Configuration Return Data With Mockito

- @WebMVCTest(<controller with .class extension>) – Indicates it is a test splice
  - With the controller it will try to bring up all controllers
- Test Code – We are testing the Beer Controller
  - In the test class
    - @Autowired mockMvc;
    - @MockBean BeerService beerService   // Provide a mock of the Beer Server since it is a dependency
    - void getBeerById() {
      - Beer testBeer = new BeerServiceImpl().listBeers().getFirst();
      -
      - // given(BeerService.getBeerById(any(UUID.class))).willReturn(testBeer);   // For any UUID return the testBeer
      - // given(BeerService.getBeerById(testBeer.id())).willReturn(testBeer);   // For the specific UUID return the test ber
      - mockMVC.perform(get(“/api/v1/beer” + testBeeer.getId())
        - .accept(MediaType.APPLICATION_JSON)
        - .andExpect(status().isOK());
        - .andExpect(contentType(MediaType.APPLICATION_JSON))
        - .andExpect(JsonPath(“$.id”), is(test.Beer.getId().toString());   // is() -- aprt of the hamcrest library
        - .andExpect(JsonPath(“$.id”), is(testBeer.getBeerName())));

Using JSON Matchers

- Jayway JSON Path – A java DSL for reading JSON documents
  - Included in the Spring Boot Test Dependencies
- Dependencies
  - <dependency>
    - <groupId>com.jayway.jsonpath</groupId>
    - <artifact>json-path</artifactId>
- Can use a dot notation or bracket notation
- JsonPath
  - Evaluate the given JsonPath expression against the response body and assert the resulting value with the given Hamcrest Matcher
  - `org.springframework.test.web.servlet.result`
- `Creating JSON using Jackson`
  - `@SpringBootTest – Use the object mapper create by Spring Boot`
    -

Creating JSON using Jackson MockMVC – Test Create Beer

- ObjectMapper – Serialize and Deserialize Data from JSON
- Use Object Mapper Created by Spring Boot -- Test your configuration
- Example
  - @AutoWired ObjectMapper objectMapper;
  - @Test
  - void testCreateNewBeer() {
    - // Create only if not using Spring Boots ObjectMapper
    - //ObjectMapper objectMapper = new ObjectMapper();
    - //objectMapper.findAndRegisterModules(); // Tells Jackson to find modules on the classpath ex. DataTimeType
    - Beer beer = beerServiceImpl.listBeers().getFirst(); beer.setId(null); beer.setVersion(null);
    - given(beerService.saveNewBeer(any(Beer.class))).willReturn(userServiceImpl.listBeers().get(1));
    -
    - mockMvc.perform(post() “/api/v1/beer”)
      - .accept(MediaType.APPLICATION_JSON)
      - .contentType(MediaType.APPLICATION_JSON)
      - ,content(ObjectMapper.writeValueAsString(beer))
      - .andExpect(isStatus().isCreated())
      - .andExpect(heaer().exists(“Location”)
  - }

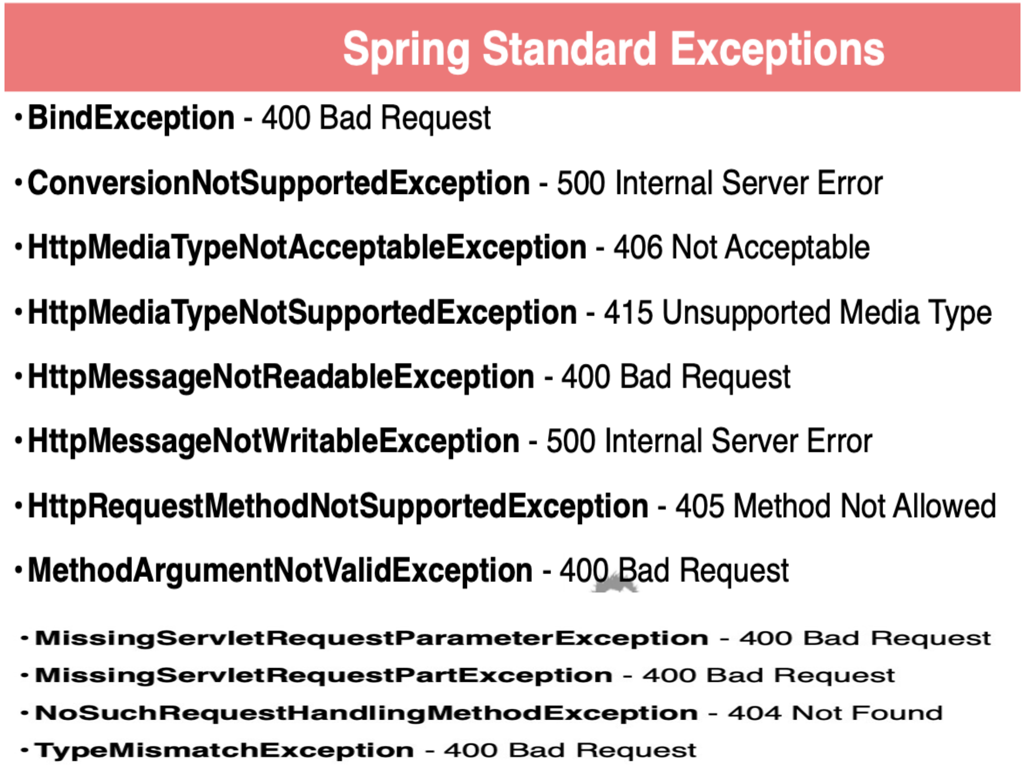
MockMVC – Update Beer

- Example
  - @Test
  - public void testUpdateBeer() throws Exception {
    - Beer beer = beerServiceImpl.listBeers().get(0);
    - mockMvc.perform(put("/api/v1/beer" + beer.getId())
      - .accept(MeditaType.APPLICATION_JSON)
      - .contentType(MediaType.APPLICATION_JSON)
      - .content(objectMapper.writeValueAsString(beer)));
      - .andExpect(status().isNoContent());
      -
    - // Verify that the parameter beer id is the actual id being sent into the function. Verifies that properties are getting sent through your code properly
    - ArgumentCaptor<UUID> uuidArgumentCaptor = ArgumentCaptorforClass(UUID.class);
    - verify(beerService).deleteById(uuidArgumentCaptor.capture());
    - assertThat(beer.getId()).isEqualTo(uuidArgumentsCaptor.getValue());
    -
    - // Verify a function has been called with the correct parameters once
    - verify(beerService).updateBeerById(any(UUID.class), any(Bear.class));
  - }

URI Builders

Exception Handling Overview

- For a 500 Series error
  - Do not leak information to the stack trace
- Spring MVC does support a number of standard exceptions
  - Spring MVC based robust support for customizing error responses
  - Handled buy the DefaultHandlerExceptionResolver
    - Does not write content to the body of the response
    - Set a status code
- Standard Spring Exceptions
- Bind Exception -- EX pass a String for a Number or Int value for a UUID
- Spring MVC Exception Handling
  - @ExceptionHandler on controller method to handle specific Exception types
  - @ResponseStatus – Annotation sets http status
  - @Controller Advice – Used to implement a global exception handler
  - ResponseStatusException Class – A thrown exception which allows setting H
  - AbstractHandlerException Resolver – full control over response including body

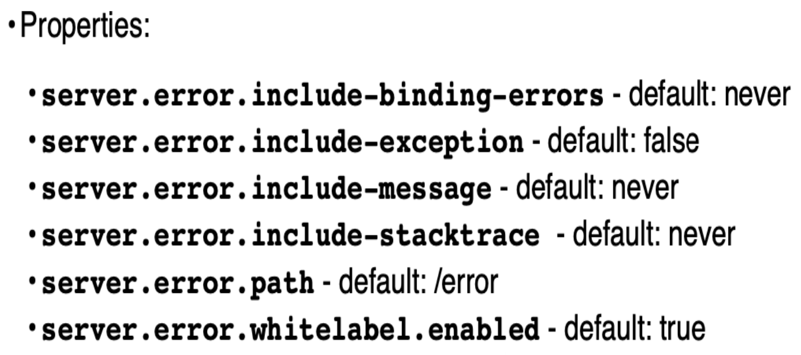


Spring Standard Exceptions

- **BindException** - 400 Bad Request
- **ConversionNotSupportedException** - 500 Internal Server Error
- **HttpMediaTypeNotAcceptableException** - 406 Not Acceptable
- **HttpMediaTypeNotSupportedException** - 415 Unsupported Media Type
- **HttpMessageNotReadableException** - 400 Bad Request
- **HttpMessageNotWritableException** - 500 Internal Server Error
- **HttpRequestMethodNotSupportedException** - 405 Method Not Allowed
- **MethodArgumentNotValidException** - 400 Bad Request
- **MissingServletRequestParameterException** - 400 Bad Request
- **MissingServletRequestPartException** - 400 Bad Request
- **NoSuchRequestHandlingMethodException** - 404 Not Found
- **TypeMismatchException** - 400 Bad Request

Spring Boot Error Control

- provides a whitelabel error page for HTML requests of JSON response for restful requests
- Srping Boot BasicErrorController – Rarely used
  - Extended for additional error response customization
  - wide support of needs of various clients and situations
  - Rarely used.
- Optional
  - Using the optional in the controller – cleaner implementation. The controller not the service decides how the exception should be returned



- Properties:
  - **`server.error.include-binding-errors`** - default: never
  - **`server.error.include-exception`** - default: false
  - **`server.error.include-message`** - default: never
  - **`server.error.include-stacktrace`** - default: never
  - **`server.error.path`** - default: /error
  - **`server.error.whitelabel.enabled`** - default: true

Exceptions : Throw Custom Exception with Mockito

- Example
  - @Test
  - public void getBeerByIdNotFound() {
    - given(beerService.gegtBeerById(any(UUID.class))).willThrow(NotFoundException.class);
    - mockMvc.perform(get(“/api/v1/beer”), UUID.randomUUID())
    - .andExpect(status().isNotFound());
  - }

Using Exception Handler Controller Advice

- Example – Inside a controller class and and NotFoundException will execute this code
  - @ExceptionHanlder(NotFoundException.class)   // Can have one or more classes
  - public ResponseEntity handleNotFoundException() {
    - return ResponseEntity.notFound().build();
  - }
- Controller Advice
  - Example
    - @ControllerAdvice   // All controllers are able to use this class
    - pubic class ExceptionController {
      - @ExceptionHanlder(NotFoundException.class)   // Can have one or more classes
      - public ResponseEntity handleNotFoundException() {
      - return ResponseEntity.notFound().build();
      - }
    - }

Exceptions – ResponseStatus

- For Rest the HttpStatus is the primary value need
- @ResponseStatus has two parameters (Http Status Code) and NotFound

Spring DTO

- Data Structures which do not have behavior, but transfer data between producers and consumers
- Why not use entities
  - Spring data Rest exposes database entities directly can be used for simple project
  - Database entities can leak data to the client tier
  - Separations – As the application become more and more complex separation become more and more important
    - The needs of the consumer are different than the need of the persistence
    - See the divergence between what the web tier needs and the persistence tier needs
  - DTO can be optimized for JSON serialization and deserialization
  - With DTO you can separate classes in the backend and send them as one Data Structure to the front end
    - Ex. Lets say you have customer and beer then you can keep separate in the backend and Create a class that combines them to send to the front end
- Type Conversions
  - Best Practice is to used dedicated converters (single Responsibility Principle)
    - service should not be doing type conversions
  - Spring Framework provides a Interface called Converter with generics which can be used with the Conversion Service
  - MapStruct is a code generator which automates generation of type converters

Spring DTO
Spring Data JPA Dependencies
Creating JPA Entities
Hibernate UUID id Generation

  - Code Generator
  - Provide the interface and MapStruct generates the code
  - Work with annotations
  - Has good spring integration can generate Spring Converters and Components
  - Inject into services

- Spring Data JPA Dependencies : org.springframework.boot:spring-boot-starter-jpa / com.h:h2database:h2
- Creating JPA Entities
  - @Version – Compare the version in the instance to the version column in the database and if different throws an exception
- Hibernate UUID
  - UUID – Infinite number of possibilities and more performant since the event space is huge
  - @GenericGenerator – Hibernate specific generation which contains one parameter name and strategy
    - @GenericGenerator(name = “UUID”, strategy = “org.hibernate.id.UUIDGenerator”)
  -

Spring Boot JPA Test Splice

- Spring Boot JPA Test Splice
  - @DataJpaTest // Autowiring a controller would cause it to fail
  - class BeerRepositoryTest {
    - @Autowired BeerRepository beerRepositoryTest;
    - @Test
    - void testSaveBeer() {
      - Beer savedBeer = beerRepository.save(Beer.builder().beerName(“My Beer)).build();
      - assertThat(savedBeer).isNotNull();
      - assertThat(savedBeer).getId()).isNotNull();
    - }
  - Testing sever items
    - will have a beer repository in the spring context
    - SpringBoot Bring up a minimal database context
      - Hibernate will do reflection on the two created entities to brings thing in
    - The H2 memory db will be initialized by hibernate
    - The injected repository will save the new beer and verify that the assertions will prove there is an id.
  - Every time a test completes then Spring will do rollback to make sure the data is in the original state

MapStruct Dependencies and Configuration

- Maven
  - declare property for org.mapstruct.version
  - dependency: org.mapstruct:mapstruct:<property version>
  - <plugin>
    - <groupId>org.apache.maven.plugins</groupId>
    - <artifactId>maven-compiler-plugin</artifactId>
    - <version>3.10.1</version>
    - <configuration>
      - <source>17</source> // Define the java versions for source and target
      - <target>17</target>
      - <annotationProcessingPaths>
        - <path>
          - <groupId>org.progject.lombok</groupId> <artifactId>lombok</artifactId> <version>${lombok.version}</lombok>
        - </path>
        - <path>
          - <groupId>org.mapstruct</groupId> <artifactId>mapstruct-processors</groupId> <version>${org.mapstruct.version}</version>
        - </path>
        - <path>
          - <groupId>org.projectlombok</groupId> <artifactId>lombok-mapstruct-bindings</groupId><version>0.2.0</version>
        - </path>
        - <compilerArgs> // Arguments for the compiler
          - <compilerArg>-Amapstruct.defaultComponentModel=spring</compilerArgs // create classes with @Component
        - </compilerArgs>
      - </annotationProcessingPaths>
    - </configuration>
    - <plugin>

MapStruct Mappers

- Create interface
  - @Mapper
  - public interface BeerMapper {
    - Beer BeerDtoToBeer(BeerDTO dto);
    - BeerDTO BeerToBeerDto(Beer beer)
  - }

JPA Services
JPA Get Operations
Controller Integration Test

- Get a list of instances
  - beerRepository.stream.map(beerMapper::beerToBeerDto).toList();
- Get by Id
  - Optinal.ofNullable(beerMapper.beerToBeerDto(beerRepository.findById(id)).orElse(return null);
- Integration Test
  - @SpringBootTest – Bring up the full context not just a splice
  - Example Problem with Integration test the data is changed permanently in order to avoid this use
    - @Rollback
    - @Transacitonal
    - @Test
    - void testEmptyList() {
      - beerRepository.deleteAll()
      - List<BeerDTO dtos = beerController.listBeers();
      - assertThat(dtos.size()).isEqualTo(0);

Save New Beer
Update a Beer

- Save an object
  - return beerMap.beerToBeerDto(beerRepository.save(beerMapper.beerDtoToBeer(beer)));
- update an object
- // Cannot do any updates outside the lambda function
- AtomicReference<Optional<BeerDTO>> atomicReference = new AtomicReference<>();
- beerRepository.findById(beerId).ifPresentOrElse(
- foundBeer -> {
- foundBeer.setBeerName(beerDTO.getBeerName()); foundBeer.setBeerStyle(beerDTO.getBeerStyle());
- foundBeer.setUpc(beerDTO.getUpc()); foundBeer.setPrice(beerDTO.getPrice());
- atomicReference.set(Optional.of(beerMapper.beerToBeerDTO(beerRepository.save(foundBeer))));
- },
- () -> { atomicReference.set(Optional.empty()); });
- return atomicReference.get();
- When checking for existing using a Repository check for existId since that will not create the POJO and waste time and resources.

Overview of Java Bean Validation

- Making assertions against data to ensure data integrity
- Should be happening at every exchange
  - User Input Data
    - Don’t call the API if the data is bad
  - Should be validated early in the controller before the service layer
  - Should be validated before going to the database layer
    - Database Constraints will enforce data validation
- Java Bean Validation (changed to Jakarta Bean Validation)
  - Java Standard which provides a standard way of performing validation and handling errors (JSR 303)
    - much more graceful than if-blocks and exceptions
    - An API
    - Primary focus was to define annotations for data validation
      - Originally -- largely field level properties, but later expanded to validate input parameters
      - Does dependency injection for Bean Validation Components
      - Use Java 8 language features

Overview of Validation

- @Null, @NotNull, @AssertTrue, @AssertFalse, @Min, @Max
- @DecimalMin, @DecimalMax, @Negative, @NegativeOrZero, @Postive, @PositiveOrZero,
- @Size (check if string or collection) is between min and max
- @Digits – Check for integer digits and fraction digits
- @Past, @PastOrPresent,@Future, @FutureOrPresent – Compare Dates
- @Pattern
- @NotEmpty – Checks if value is null or empty
- @NonBlank – Checks String is not null or whitespace characters
- @Email
-
-

Overview of Validation

- Hibernate does have some specific Constraints
  - @ScirptAssert class level annotation checks class against script
  - @CreditCardNumber
  - @Currency
  - @DurationMax – Duration less than given value
  - @DurationMin – Duration greater than given value
  - @EAN –Valid EAN Barcode
  - @ISBN valid ISBN value
  - @Length – String length between a min and max
  - @CodePointLength – Validate that code point length of the annotated character sequence is between min and max included
  - @Luhncheck – Luhn check sum
  - @Mod10Check – Mod 10 Check sum
  - @Mod11Check – Mod 11 Check sum
  - @Range – checks if number is between given min and max inclusive
  - @SafeHtml
  - @UniqueElements
  - @Url

Overview of Validation

- Validation and SpringFramework
  - Validation support can be used in controllers and services and other Spring Managed Component
  - Spring MVC will return a 400 (Bad Request) Error for validation features
  - Spring Data JPA will throw an exception for JPA constraint violations
  - SpringBoot will autoconfigure validation when the validation implementation is found in the classpath
    - If API is only on classpath (with no implementation) you can use the annotations, but validation will not happen

Java Bean Validation Maven Dependencies Controller Binding Validation Custom Validation Handler

- Dependencies
  - org.springframework.boot:spring-boot-starter-validation
- Controller Binding Validation
  - In the Beer DTO
    - @NotNull
    - @NotBlank
    -
    - private String beerName;
  - In the BeerController add the @Validate to alert the framework that validation needs to be done.
    - public ResponseEntity handlePost(@Validated @RequestBody BeerDTO beer)
- Custom Validation Hanlder
  - Can set mockMvc.perform returns a MockSet which has its own functions

JPA Validation
Database Constraint Validation

- For the repository tests
  - Without this function, the test will pass even though there are constraints issues.
    - Problem: It running so quickly, but the session is ending to quickly
    - Solution: Use beerRepository.flush() to immediately write to disk
  - Put the validations in both the DTO and the Model so that different layers can validate (ex. service layer can validate when a beer has been patched)
- Best Practice : Don’t hit the database unless all the data is validated
- Database Constraint Validation
  - Default Behavior of hibernate set string properties to 255
  - How do we change the size of the String
    - Use @Column(length = 50)
    - @Size(max=50) would even be better since we don’t try to save data to the database unless it has been validated.
-

Controller Testing with JPA

- Controller Testing with JPA
  - Example
    - @AutoWired WebApplicationContext wap;
    - Mockmvc mockMvc;
    - @BeforeEach0
      - void setUp() { mockMvc = MockMvcBuilders.webAppContextSetup(wac).build()
        - Setup the Spring Web Application environment with the Data Repositories which is full Spring Boot Test
  - For this test we are using the Patch
    - However patch requires a Hash Table for the maps
    - @Validated will not be used with Hashtables so at the controller level the changes will not be validated
      - In the current solution the changes to the Beer Instance will not be validated
      - Which means saving the object could cause a constraint problem in the database.
      -

JPA Validation Error Message

JPA Validation Error Handler

Annotations -- Database

- @Entity
- @id
  - @GeneratedType The strategy used to generate the unique key
  - @GeneratedValue(generator="UUID")
  - @GenericGenerator(name = "UUID", strategy = "org.hibernate.id.UUIDGenerator")
- @Jointable Which Tables to join
  - @Join Column the column that will do the joining
- @Version – Part of the locking Strategy where every update is incremented by 1.
  - If the instance value is different from the database then some other application or thread overwrote it.
- @Column(length = 36, columnDefinition = "varchar", updatable = false, nullable = false)
- @Transaciotnal, @Rollback

Spring and Spring Boot Annotations

- @autowired, @Primary, @Qualifier, @Profile, @ActiveProfile, @Component, @Bean
- @Controller, @Service, @Component, @Repository, @RestController
  - @RestController – returns responseBody -- object returned is automatically serialized into JSON and passed back into the HttpResponse object.
- @RequestMapping, @PathVariable(name of Path Variable), @ResponseBody
  - @RequestBody – Deserializes the HTTP Request Body into a Data Transfer Object.
- @GetMapping, @PostMapping, @PutMapping, @DeleteMapping, @PatchMapping
- @ExceptionHandler, @ControllerAdvice, @ResponsseStatus
  - @ResponseStatus(value = HttpStatus.NOT_FOUND, reason="The primary key was found in the data structure or database")
- @DataJpaTest, @SpringBootTest, @WebMvcTest
  - @SpringBootTest – Used for integration test (Testing the controller as if it was the spring framework, but don’t have the webcontex
  - @DataJpaTest, @WebMvcTest – Used for unit testing.
- @Validated (Can be on a parameter or field), @NotBlank, @NotNull

Others

- MapStruct
  - @Mapper
  - When the compile task is execute mapstrcut will generated the code found in the interfaces annotated with @Mapper
  - create functions that can convert from DTO and to DTO
- ObjectMapper

Mockito Annotations and Extra

- @MockMvc
- @MockBean
- @Captor
- given().willRetrun()
- verify(Interface).delete(argment.capture())
  - uuidArgumentCaptor.getValue()
- mock.perform(get(<uri>, path Variables)
  - .accept
  - .andExpect status.isOk(), contentType(), header.exists(“location”)
    - jsonPath($.metadata.id”, is(cusotmer.getMetaData().getId())
  - For adding content: contentType, content
    - ObjectMapper.writeValueAsStriong