Spring Boot Notes

# What is it?

A framework module that provides **Rapid Action Development** to the Spring framework. It is a configuration management tool.

## Spring Boot’s Improvements to Spring

Native Component Scan for annotations (@Service, @Repository).

# Spring Framework

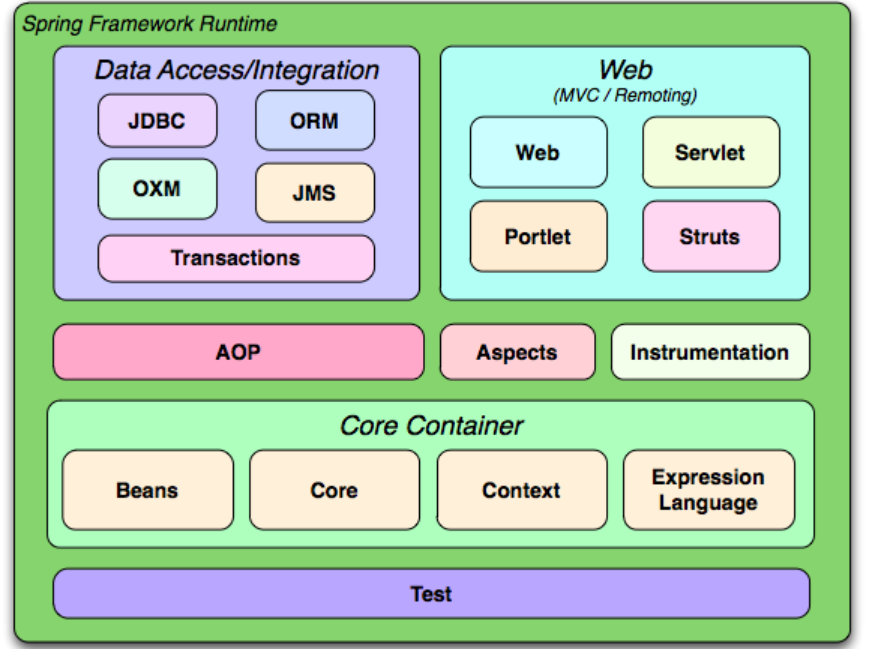
Provides comprehensive infrastructure support for developing Java applications. It allows app building from “plain old Java objects” (POJOs). It allows Java methods to interface with multiple APIs without using their transaction syntax.

## Dependency Injection

A technique where one object supplies the dependencies of another; the dependency (service) is passed into an object (the client) and made part of the client’s state. This is more efficient than making the client build or find the service. This is an **Inversion of Control** in that the framework takes care of service integration.

DI/IoC allows loose coupling of applications = easy testing.

## Features of Spring Framework



* Servlet contains model-view-controller

# Spring MVC



(Model-View-Controller) – designed around a DispatcherServlet that dispatches requests to handlers. Spring’s has configurable handler mappings, view resolution, locale and theme resolution & file upload support.

The Servlet dispatches requests to controllers

# Annotations

**Key:**

Class level = above the class declaration

Method level = above the method declaration

Parameter level = included inside parameter parentheses before the parameters themselves

“ –“ = no parameters

Spring Boot Web Starter

Spring JPA

Misc or Unknown

Swagger 2 for Spring

Mockito JUnit testing

|  |  |  |  |
| --- | --- | --- | --- |
| Annotation | Parameters | Location | Description |
| @ComponentScan(basePackages = {"",""}) | Main SpringBootApplication package name (e.g. com.mintree.demo) and external controller package name (e.g. com.mindtree.demo.supercontroller) | Class level; above main application class | Allows controllers outside the main app package to be found & utilized  🡪Not required of Spring Boot as much as Spring framework itself |
| @Controller |  | Class level | Defines a controller |
| @EnableSwagger2 |  |  |  |
| @GetMapping, @PostMapping, @DeleteMapping | Path string | Method level | Sets path mapping for model view of method for GET, POST, DELETE HTTP methods |
| @PathVariable | Method parameter | Method level; parameter declaration | binds a path variable with a method parameter |
| @Repository | - | Interface level | Define an Interface that extends JpaRepository to provide default implementation for CRUD operations  🡪 Use on the DAO layer interfaces to indicate this is as such |
| @RequestMapping | Path string | Class level | Sets path mapping for all model views of controller class |
| @ResponseBody |  | Method level; parameter declaration | Indicates that the return value of a method should be used as the response body of a request |
| @RestController | - | Class level | Combines @Controller and @ResponseBody to make web requests that returns data |
| @SpringBootApplication | - | Class level | Combines @Configuration, @EnableAutoConfiguration, @ComponentScan. |
| @Valid |  | Method level; Parameter declaration | Ensures that request body is valid (e.g. not blank) |
| @Value |  |  | Injecting values into fields in Spring-managed beans; expression-driven dependency injection |
| @Autowired | - | Property level | Allows Spring to resolve & autoinject collaborating dependencies into your bean  🡪Use on properties of the service or client layer (ServiceImpl classes that use Repos, Controller classes that use ServiceImpl classes) to let Spring Boot Component Scan work properly |
| @Column | e.g. (nullable = false, updatable = false); name, length, nullable, updateable etc. | Property level | Define properties of column to be mapped to the annotated field |
| @CrossOrigin | origins = localhost:4200 | Class (Entity) level | Allow backend on port 8080 to communicate to frontend on port 4200 i.e. “cross-origin resource sharing” |
| @Entity | Optional name=”TableName” for explicit naming | Class level | Marks class as a persistent Java class |
| @EntityListeners | e.g. AuditingEntityListener class |  |  |
| @GeneratedValue | e.g. (strategy = GenerationType.IDENTITY) | Property level | Defines primary key auto generation strategy |
| @JoinColumn | Foreign key (e.g. name = “iddepartment”)  name, joinColumns, inverseJoinColumns | Property level | specifies the name of column for foreign key which relates owner class to owned class in the database |
| @JoinTable | name, joinColumns, inverseJoinColumns | Property level | Specify table name and names of foreign keys to be mapped together |
| @Id | - | Property level | Defines primary key of Entity |
| @Max, @Min | value | Property level | Define the max and min value constraints for a given integer |
| @NotBlank | - | Property level | Validates that annotated field is not null or empty |
| @NotNull | - | Property level | Ensures that a given property may not be provided as null to the database |
| @OneToOne, @OneToMany, @ManyToOne, @ManytoMany | mappedBy(= “name of the association-mapping attribute”) | Property level | See “Entity Association Mapping” |
| @Query | SQL Query String | Method level | Define a specific SQL statment |
| @RunWith | e.g. SpringRunner.class |  |  |
| @Service | - | Class level | 🡪 Use on the service layer implementation to indicate that the following class is such |
| @Size | max, min | Property level | Specifies the length of a String property |
| @Table | (name = “TableName”) | Class level | Provides details of table Entity will be mapped to |
| @Temporal |  | Property level | Used w/java.sql.Date; converts date & time values from Java Object to compatible database type and vice versa |
| @Test | - | Method level |  |
| @JsonIgnoreProperties | e.g. (value = {"createdAt", "updatedAt"}, allowGetters = true) | Class level | Jackson annotation; used for serializing & deserializing to json |
| @EnableSwagger2 | - | Class level | Enables Swagger 2 (for creating easy Spring API documentation) |
| @Before | - | Property level |  |
| @InjectMocks |  |  |  |
| @Mock |  |  |  |

# Running the App

Ctrl + F11 or Run as Spring Boot Application, then go to localhost:8080 + your mapping

## Running as Live Server

Insert the following Maven dependency:

<dependency>

    <groupId>org.springframework.boot</groupId>

    <artifactId>spring-boot-devtools</artifactId>

</dependency>

# Hibernate

An ORM (Object-Relational Mapping) for Java supported by Spring Boot. It uses an association table to map the relationship between constructed entities.

Tutorial(“Hibernate Example using Annotation in Eclipse”): <https://www.javatpoint.com/hibernate-with-annotation>

Tutorial (“Spring Boot, MySQL, JPA, Hibernate Restful CRUD API Tutorial”): <https://www.callicoder.com/spring-boot-rest-api-tutorial-with-mysql-jpa-hibernate/>

Best practices: <https://docs.jboss.org/hibernate/core/3.3/reference/en/html/best-practices.html>

* By default, Hibernate will generate table names in lowercase letters
* By default, a field named createdAt is mapped to a column named created\_at in the database table. i.e. all camel cases are replaced with underscores
* org.hibernate import is depreciated; use javax.persistence
* changing the database schema will ruin the current configuration/mapping

## Config

Create file to add as resource to java folder – hibernate.cfg.xml:

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 5.3//EN"

"http://www.hibernate.org/dtd/hibernate-configuration-5.3.dtd">

<hibernate-configuration>

<session-factory>

<property name=*"hbm2ddl.auto"*>update</property>

<property name=*"dialect"*>org.hibernate.dialect.MySQLDialect</property>

<property name=*"connection.url"*>jdbc:mysql://localhost:3306/DBname</property>

<property name=*"connection.username"*>root</property>

<property name=*"connection.password"*>Welcome123</property>

<property name=*"connection.driver\_class"*>com.mysql.jdbc.Driver</property>

<!-- <mapping class=*"com.mindtree.AppName.EntityName"*/> -->

</session-factory>

</hibernate-configuration>

## Entity

Class names will now be plotted with “@Entity” so as to let hibernate know it must use that template to implement the database. (Needed **Maven dependency** – javax.persistence)

## Entity Association Mapping

Tutorial: <http://www.thejavageek.com/2014/01/15/jpa-many-one-association/> ; <https://thoughts-on-java.org/ultimate-guide-association-mappings-jpa-hibernate/>

Ask – which table has ownership of the other i.e. what table contains the foreign key of the other?

If there are many of the owner class for the owned class (e.g. many Employees for one Department), add a @ManyToOne annotation to the declaration of the owned class in the owner class :

*@Entity*

public class Employee {

*@TableGenerator*(name = "employee\_gen", table = "id\_gen", pkColumnName = "gen\_name", valueColumnName = "gen\_val", allocationSize = 1)

*@Id*

*@GeneratedValue*(strategy = GenerationType.TABLE, generator = "employee\_gen")

private int idemployee;

private String firstname;

private String lastname;

private String email;

*@ManyToOne*

*@JoinColumn*(name = "iddepartment")

private Department department;

**Unidirectional vs. Bidirectional:** A bidirectional relationship has both an owning side and an inverse side. A unidirectional relationship has only an owning side. The owning side of a relationship determines how the Persistence runtime makes updates to the relationship in the database.

### Unidirectional

#### Unidirectional Many-to-One

Remember, the side of the mapping with the foreign key is the owning side and gets the @JoinColumn annotation.

#### Unidirectional One-to-Many

This is rarely used but may be used to implement a list of the owned class in the owner class.

### Bidirectional

#### Bidirectional Many-to-One

Consists of 2 parts: the owning part (to-many) and the part that references the mapping (to-one).

Very similar in function to Unidirectional Many-To-One although any update to one needs to occur on the other as well (insertion, deletion).

* Because this is error-prone, try to make methods that add/delete both at once

### Many-to-Many Associations

Involves making another table of primary key associations typically.

E.g. Product & Stores – each Store sells multiple Products and each Product gets sold in multiple Stores.

🡺 Use a Set instead of a list as the attribute type else Hibernate will run inefficiently.

Requires both joinColumns & inverseJoinColumns in the @JoinTable annotation. The joinColumns attribute defines the foreign key columns for the entity on which you define the association mapping. The inverseJoinColumns attribute specifies the foreign key columns of the associated entity.

## Revised Association Mapping

### One-To-One

### One-To-Many

### Many-To-One

### Many-To-Many

#### Bidirectional

Relationship definition on owning side has 2 mandatory & 1 optional parts:

* mandatory @Many-To-Many annotation
* mandatory List of owned attribute
* optional @JoinTable annotation
  + define name of join table and foreign key columns

## SessionFactory

Instantiate only one SessionFactory per program.

## Session

Instantiate a session each time an interaction is needed with the database.

## Transaction

# Spring JPA

## Config

Does NOT require Hibernate config

Change the schema name in the url

## Spring DATASOURCE (DataSourceAutoConfiguration & DataSourceProperties)

spring.datasource.driver-class-name=com.mysql.jdbc.Driver

spring.datasource.url=jdbc:mysql://localhost:3306/city\_app?useSSL=false

spring.datasource.username = root

spring.datasource.password = Welcome123

## Hibernate Properties

# The SQL dialect makes Hibernate generate better SQL for the chosen database

spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5Dialect

# Hibernate ddl auto (create, create-drop, validate, update)

spring.jpa.hibernate.ddl-auto = update

## SimpleJpaRepository

Default implementation of JpaRepository interface which defines CRUD operations.

Has methods: save(), findOne(), findAll(), count(), delete() etc.

* repositoryName.findAll() is like SELECT \* FROM repositoryName;
* findOne() is depreciated; use getOne()to get by id

Since there is the SimpleJpaRepository default implementation, there is no need to implement the above methods.

* Documentation: <https://docs.spring.io/autorepo/docs/spring-data-jpa/current/api/org/springframework/data/jpa/repository/support/SimpleJpaRepository.html>

## Practices

* Do not use @Controller for a Controller acting as a REST API rather than a view Controller
* Use an @JoinTable if you’d like the history of all entity associations (e.g. returning the history of Reservations for Customer at a given Hotel)
* @JsonIgnore is useful for preventing infinite loops in bidirectional mappings
  + In the future, this will not happen due to use of DTOs
* Actions (POST, PUT, DELETE) should not be exposed to the end user.
  + We can use the same mapping with different REST operations (e.g. /hotels/1 can be a @GetMapping and it can be an @PostMapping)
* To add dummy data, keep a SQL query (INSERT INTO table VALUES) ready in Query Browser to add the data in

## Exceptions

It is important to define the HTTP status code to be thrown when an exception is thrown by your application.

### Frequent Exceptions

DataAccessException

ConstraintViolationException

ResourceNotFoundException will be thrown when Entity with given Id is not found in database when created with @ResponseStatus annotation.

# Postman Validation

Postman may be used by backend developers to validate that their REST apis work without needing a running frontend.

## PUT Operations

Using the “Body” tab, write a JSON object with the properly defined syntax (Keys should reflect what you coded as properties of objects). Validate that the return message is what has been defined.

# JSR Specification

All REST methods should have status, message, return

GetMapping methods that return List should return Map<String, ?> (e.g. getAllTracks())

Remember to always have a default constructor so that findAll doesn’t throw an error

# Easy API Documentation

Possible using Swagger 2: <https://www.baeldung.com/swagger-2-documentation-for-spring-rest-api>

Place SwaggerConfig class at same application level as the Spring Boot Application.

# Connecting to the Front-end

JSP – a rendering engine. JavaScript will only be required when you want interaction with the client-side (DOM manipulation, change aesthetic); anything you’d like to change without a refresh.

1. Add the Maven dependency: org.apache.tomcat.embed (artifactID: tomcat-embed-jasper)
2. Add to application properties:

spring.mvc.view.prefix=/WEB-INF/view

spring.mvc.view.suffix=.jsp

1. ? the folder in the same level as “java”: webapp -> WEB-INF -> view
   1. Write HTML files here as .jsp files e.g. index.jsp
2. Must write a Controller for view to work! Have @Controller return .jsp file name w/o .jsp e.g. return “index”:

*@Controller*

public class SampleController {

*@RequestMapping(“/”)* 🡨 THE PATH

public String hello(ModelMap map) {

return “index”; 🡨 THE PAGE

}

}

1. 3 things you can pass into controller:
   1. ModelMap
      1. ${message} in .jsp file – give model attributes; need to specify modelMap.addAttribute(“message”, “Welcome to MVC!”) to display “Welcome to MVC!” where you had ${message} in the .jsp file
         1. The browser will not see this! It will render the content as normal html.
   2. ModelView
   3. Model

## Bootstrap

Implement the Maven dependency. Add the bootstrap link into your jsp fie in the same way you’d add a CSS link to an HTML file.

Bootstrap has a container class making everything in the center of a page.

### Adding External CSS Files for More Custom Input

Spring will component scan the “static” folder for a folder called “css”; this is where to place your stylesheet. Then, link to the stylesheet in your jsp file using “css/myStyleSheetName.css”.

## Practices

* Can pass Java mapped paths into href to navigate to new pages of server
* After POST, redirect to a GET page using return “redirect:/pagepath”
* DO NOT write logic in jsp files; that is for the backend.

# JSTL – Java Standard Tag Library

A collection of JSP tags supporting common structural tasks such as iteration, conditionals, or SQL tags.

# DTO – Data Transfer Object

In the future, instead of passing entities to methods, we will pass DTOs. Entities and DTOs will be mapped using ModelMapper.

# ControllerAdvice Exception Handling

Tutorial:

<https://www.javainuse.com/spring/boot-exception-handling>

ControllerAdvice by default applies to all classes that use the @Controller annotation (and therefore @RestController).