

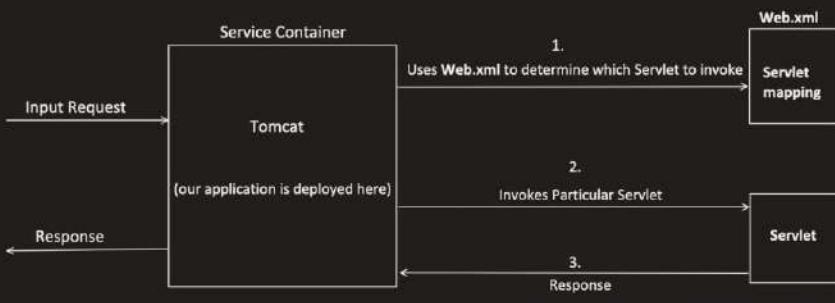
## Table of Contents

Spring boot : Introduction .....	2
Spring boot: Project Setup and Layered Architecture (Concept && Coding) .....	10
SpringBoot: Maven.....	11
SpringBoot - Annotations (Controller Layer) Part1 .....	21
Spring boot : Bean and its Lifecycle .....	24
Springboot: Dependency Injection .....	33
Springboot: Bean Scopes by Concept && Coding.....	43
Springboot: Dynamic Bean Initialization .....	51
Spring boot: @Profile.....	55
Spring boot: @ConditionalOnProperty.....	61
Spring boot : AOP .....	63
Spring boot: @Transactional (Part1).....	76
Spring boot: @Transactional (Part-2).....	83
Spring boot - @Transaction Part3 (Isolation Level).....	91
Spring boot - Async Annotation (Part1) .....	94
SpringBoot @Async Annotation - Part2.....	104
Springboot: Custom Interceptors.....	111
Springboot: Filters Vs Interceptors.....	117
Springboot: Filters Vs Interceptors.....	121
Spring boot: HATEOAS.....	125
Spring boot: ResponseEntity and Codes .....	131
Springboot : Exception Handling.....	138
Spring boot: JPA (Part-1) .....	153
JPA ARCHITECTURE (PART2) .....	158
Springboot: First Level Caching .....	168
JPA (PART4) - Second Level Caching .....	173
JPA - PART5 (DTO - TABLE) .....	184
OneToOne Mapping (Unidirectional and Bidirectional)Part 6 .....	194
JPA-Part7 .....	255

## Spring boot : Introduction

Before we talk about Spring or Spring Boot, first we need to understand about "SERVLET" and "Servlet Container"

- Provide foundation for building web applications.
- Servlet is a Java Class, which handles client request, process it and return the response.
- And Servlet Container are the ones which manages the Servlets.



**Servlet1:**

```
@WebServlet("/demoservletone")
public class DemoServlet1 extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response) {
        String requestPathInfo = request.getPathInfo();
        if(requestPathInfo.equals("/")) {
            //do something
        } else if(requestPathInfo.equals("/firstendpoint")) {
            //do something
        } else if(requestPathInfo.equals("/secondendpoint")) {
            //do something
        }
    }

    @Override
    protected void doPut(HttpServletRequest request,
                         HttpServletResponse response) {
        //do something
    }
}
```

**Web.xml**

```
<!-- my first servlet configuration below-->
<servlet>
    <servlet-name>DemoServlet1</servlet-name>
    <servlet-class>DemoServlet1</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>DemoServlet1</servlet-name>
    <url-pattern>/demoservletone</url-pattern>
    <url-pattern>/demoservletone/firstendpoint</url-pattern>
    <url-pattern>/demoservletone/secondendpoint</url-pattern>
</servlet-mapping>

<!-- my second servlet configuration below-->
<servlet>
    <servlet-name>DemoServlet2</servlet-name>
    <servlet-class>DemoServlet2</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>DemoServlet2</servlet-name>
    <url-pattern>/demoservletwo</url-pattern>
</servlet-mapping>
```

**Servlet2:**

```
@WebServlet("/demoservletwo/*")
public class DemoServlet2 extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response) {
        //do something
    }

    @Override
    protected void doPut(HttpServletRequest request,
                         HttpServletResponse response) {
        //do something
    }
}
```

**Spring Framework**, solve challenges which exists with Servlets.

- Removal of web.xml
  - this web.xml over the time becomes too big and becomes very difficult to manage and understand.
  - Spring framework introduced Annotations based configuration.
- Inversion of Control (IoC)
  - Servlets depends on Servlet container to create object and maintain its lifecycle.
  - IoC is more flexible way to manage object dependencies and its lifecycle (through Dependency injection)
- Unit Testing is much harder
  - As the object creation depends on Servlet container, mocking is not easy. Which makes Unit testing process harder.
  - Spring dependency injection facility makes the Unit testing very easy.
- Difficult to manage REST APIs
  - Handling different HTTP methods, request parameters, path mapping make code little difficult to understand.
  - Spring MVC provides an organised approach to handle the requests and its easy to build RESTful APIs.

There are many other areas where Spring framework makes developer life easy such as : integration with other technology like hibernate, adding security etc...

The most important feature of Spring framework is **DEPENDENCY INJECTION** or **Inversion of Control (IoC)**

Lets see an example **without** Dependency Injection:

```
public class Payment {  
  
    User sender = new User();  
  
    void getSenderDetails(String userID){  
        sender.getUserDetails(userID);  
    }  
}
```

```
public class User {  
  
    public void getUserDetails(String id) {  
        //do something  
    }  
}
```

Payment class is creating an instance of User class, and there is one Major problems with this and i.e.

**Tight coupling:** Now payment class is tightly coupled with User class.

**How?**

-> Suppose I want to write Unit test cases for Payment "getSenderDetails()" method, but now I can not easily MOCK "User" object, as Payment class is creating new object of User, so it will invoke the method of User class too.

-> Suppose in future, we have different types of User like "admin", "Member" etc., then with this logic, I can not change the user dynamically.

Now, Lets see an example with Dependency Injection:

```
@Component  
public class Payment {  
  
    @Autowired  
    User sender;  
  
    void getSenderDetails(String userID){  
        sender.getUserDetails(userID);  
    }  
}
```

```
@Component  
public class User {  
  
    public void getUserDetails(String id) {  
        //do something  
    }  
}
```

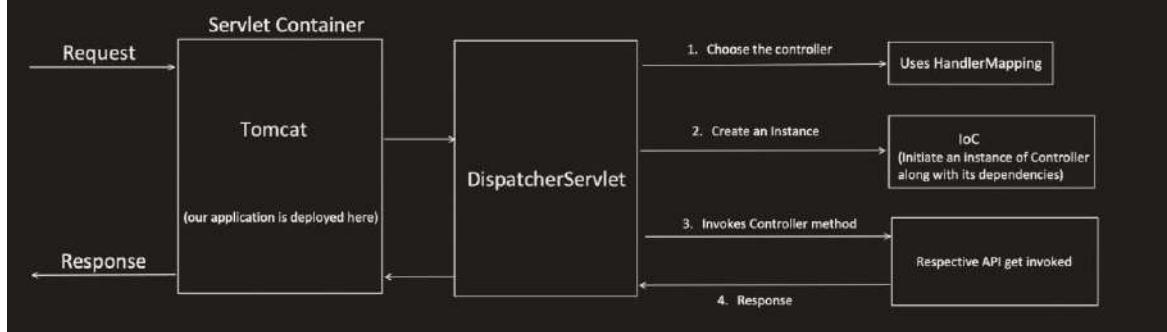
**@Component**: tells Spring that, you have to manage this class or bean.

**@Autowired**: tells Spring to resolve and add this object dependency.

The another important feature of Spring framework is lot of **INTEGRATION** available with other frameworks.

This allow Developers to choose different combination of technologies and framework which best fits their requirements like:

- Integration with Unit testing framework like Junit or Mockito.
- Integration with Data Access framework like Hibernate, JDBC, JPA etc.
- Integration with Asynchronous programming.
- Similar way, it has different integration available for:
  - Caching
  - Messaging
  - Security etc.



```
pom.xml

<modelVersion>4.0.0</modelVersion>
<groupId>com.conceptandcoding</groupId>
<artifactId>learningspringboot</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>springboot application</name>
<description>project for learning springboot</description>
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>6.1.4</version>
    </dependency>
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax-servlet-api</artifactId>
        <version>2.5</version>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.13.2</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Controller class

```
@Controller
@RequestMapping("/paymentapi")
public class PaymentController {

    @Autowired
    PaymentDAO paymentService;

    @GetMapping("/payment")
    public String getPaymentDetails() {
        return paymentService.getDetails();
    }
}
```

Config class

```
@Configuration
@EnableWebMvc
@ComponentScan(basePackages = "com.conceptandcoding")
public class AppConfig {
    // add configuration here if required
}
```

Dispatcher Servlet class

```
public class MyApplicationInitializer extends
AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return null;
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class[]{AppConfig.class};
    }

    @Override
    protected String[] getServletMappings() {
        return new String[]{"/"};
    }
}
```

**Spring Boot**, solve challenges which exists with Spring MVC.

1. Dependency Management: No need for adding different dependencies separately and also their compatible version headache.

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.2.3</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>
<groupId>com.conceptandcoding</groupId>
<artifactId>learningspringboot</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>springboot application</name>
<description>project for learning springboot</description>
<properties>
    <java.version>17</java.version>
</properties>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

2. Auto Configuration : No need for separately configuring "DispatcherServlet", "AppConfig", "EnableWebMvc", "ComponentScan". Spring boot add internally by-default.

```
@SpringBootApplication
public class SpringbootApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringbootApplication.class, args);
    }
}
```

### 3. Embedded Server:

In traditional Spring MVC application, we need to build a WAR file, which is a packaged file containing your application's classes, JSP pages, configuration files, and dependencies. Then we need to deploy this WAR file to a servlet container like Tomcat.

But in Spring boot, Servlet container is already embedded, we don't have to do all this stuff. Just run the application, that's all.

### So, what is Spring boot?

- It provides a quick way to create a production ready application.

- It is based on Spring framework.

- It support "**Convention over Configuration**".

Use default values for configuration, and if developer don't want to go with convention(the way something is done), they can override it.

- It also help to run an application as quick as possible.

pom.xml

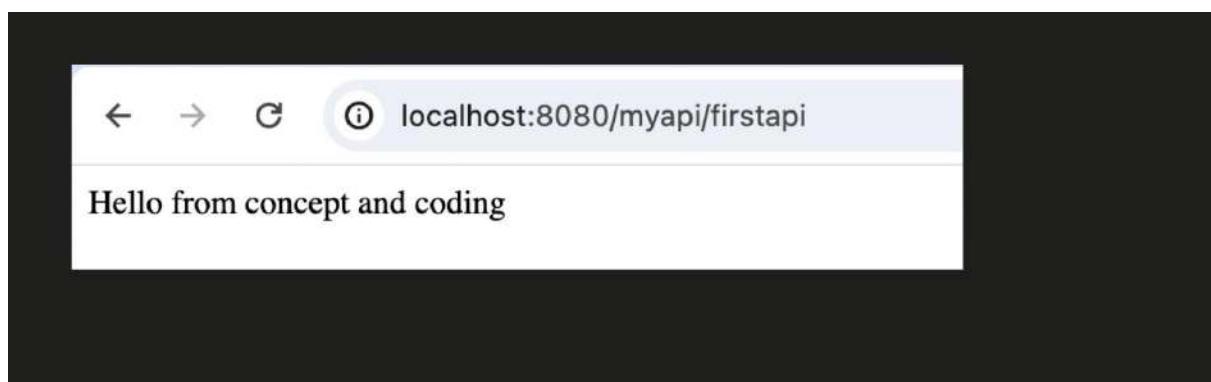
```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.2.3</version>
    <relativePath/> <!-- Lookup parent from repository --&gt;
&lt;/parent&gt;
&lt;groupId&gt;com.conceptandcoding&lt;/groupId&gt;
&lt;artifactId&gt;learningSpringboot&lt;/artifactId&gt;
&lt;version&gt;0.0.1-SNAPSHOT&lt;/version&gt;
&lt;name&gt;springboot application&lt;/name&gt;
&lt;description&gt;project for learning springboot&lt;/description&gt;
&lt;properties&gt;
    &lt;java.version&gt;17&lt;/java.version&gt;
&lt;/properties&gt;
&lt;dependencies&gt;
    &lt;dependency&gt;
        &lt;groupId&gt;org.springframework.boot&lt;/groupId&gt;
        &lt;artifactId&gt;spring-boot-starter-web&lt;/artifactId&gt;
    &lt;/dependency&gt;
    &lt;dependency&gt;
        &lt;groupId&gt;org.springframework.boot&lt;/groupId&gt;
        &lt;artifactId&gt;spring-boot-starter-test&lt;/artifactId&gt;
        &lt;scope&gt;test&lt;/scope&gt;
    &lt;/dependency&gt;
&lt;/dependencies&gt;
&lt;build&gt;
    &lt;plugins&gt;
        &lt;plugin&gt;
            &lt;groupId&gt;org.springframework.boot&lt;/groupId&gt;
            &lt;artifactId&gt;spring-boot-maven-plugin&lt;/artifactId&gt;
        &lt;/plugin&gt;
    &lt;/plugins&gt;
&lt;/build&gt;
</pre>
```

```
@SpringBootApplication
public class SpringbootApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringbootApplication.class, args);
    }
}
```

```
@RestController
@RequestMapping("/myapi")
public class MyController {

    @GetMapping("/firstapi")
    public String getData() {
        return "Hello from concept and coding";
    }
}
```

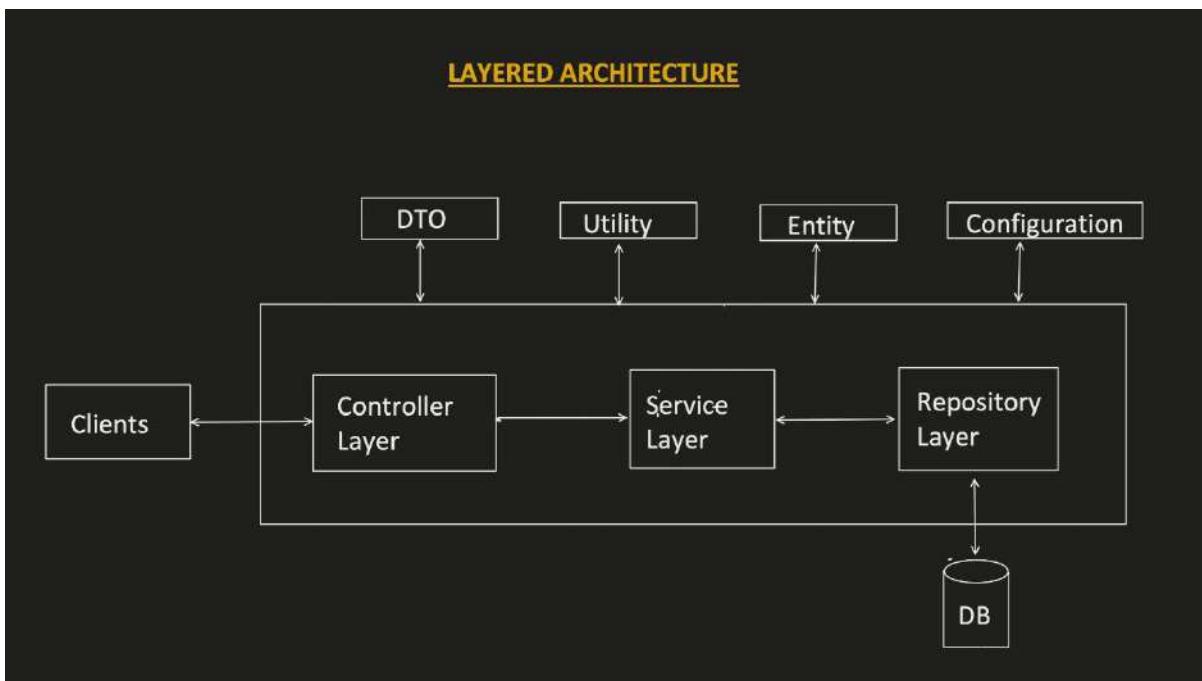


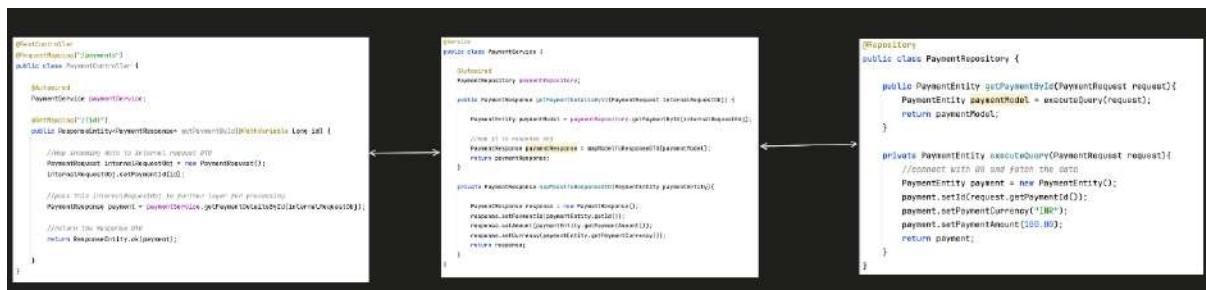
# Spring boot: Project Setup and Layered Architecture (Concept && Coding)

## Setting up the Project

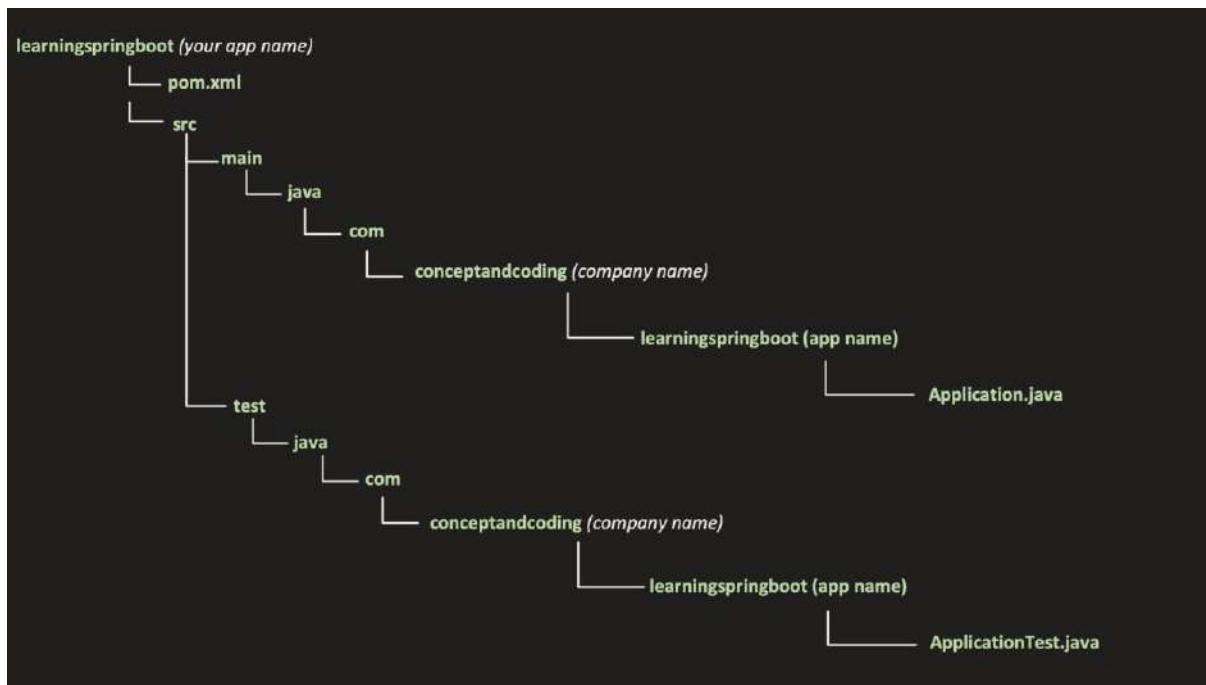
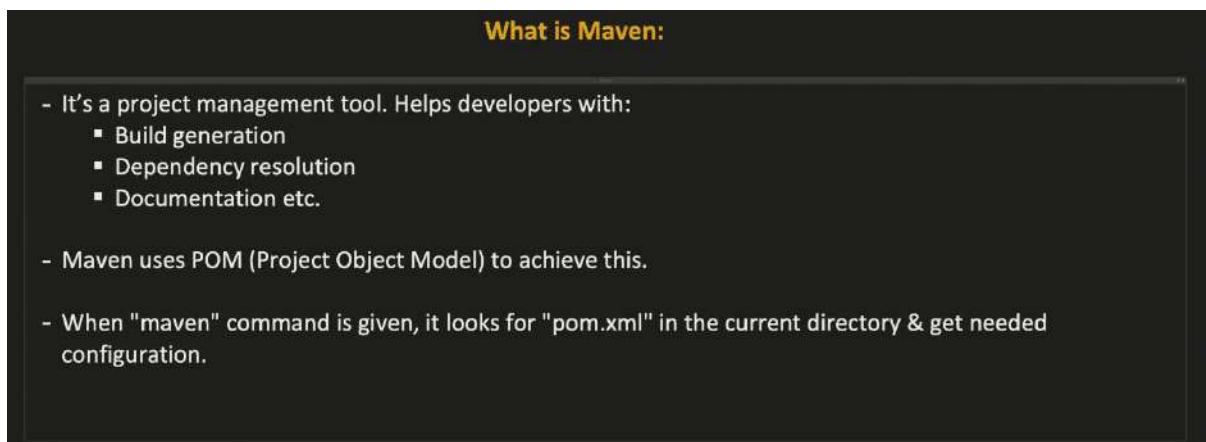
1. Go to Spring Initializr i.e. "start.spring.io"

The screenshot shows the Spring Initializr web application. It has sections for Project (Gradle - Groovy, Gradle - Kotlin, Maven), Language (Java, Kotlin, Groovy, Maven selected), and Spring Boot (3.3.0 (SNAPSHOT), 3.3.0 (M1), 3.2.4 (SNAPSHOT), 3.2.3, 3.1.10 (SNAPSHOT), 3.1.9). The 'Dependencies' section is expanded, showing 'Spring Web' (selected) and 'WEB'. Below it, there's a note: 'Build web, including RESTful applications using Spring MVC. Uses Apache Tomcat as the default embedded container.' A large 'ADD DEPENDENCIES...' button is at the top right. On the left, there's a 'Project Metadata' section with fields for Group (com.conceptandcoding), Artifact (learningspringboot), Name (springboot.application), Description (project for learning spring boot), Package name (com.conceptandcoding.learningspringboot), Packaging (Jar selected), and Java version (21 selected). At the bottom, there's a 'Java' section with a count of 17.





## SpringBoot: Maven



```

<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 https://www.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.2.3-M1</version>
    <relativePath/>
  </parent>
  <groupId>com.example.competitivedata</groupId>
  <artifactId>CompetitiveDataApplication</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <description>Competitive Data Application</description>
  <properties>
    <java.version>17</java.version>
  </properties>
  <repositories>
    <repository>
      <id>mavenCentral</id>
      <url>https://repo.maven.apache.org/maven2</url>
    </repository>
  </repositories>
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
      </dependency>
      <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
      </dependency>
    </dependencies>
  </dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>
</project>

```

Used to define the Parent project. Current project inherit the configurations from the specified Parent project. Which in turn might get inherited from the Super POM.

If this `parent` field is not specified, maven by-default inherit the configurations from "Super POM". This is the link of maven Super POM:  
<https://maven.apache.org/ref/3.2.3/maven-model-builder/super-pom.html>

Unique identifier of your project.

Define Key-Value pair for configuration. Can be referenced through out the pom file.

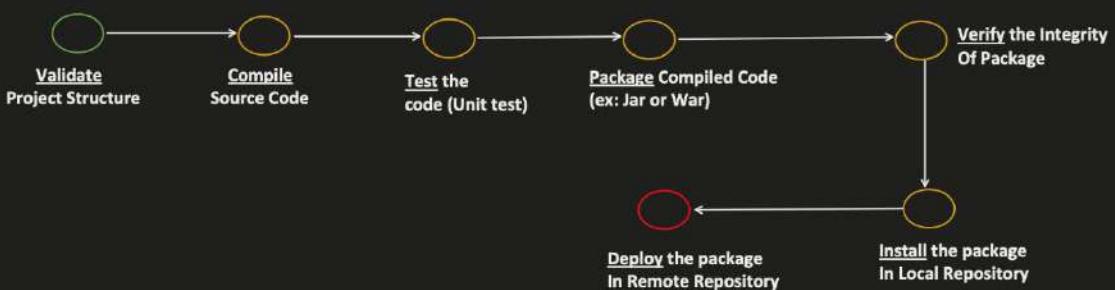
This is from where Maven look for project dependencies and download the artifacts (JARs).

This is where we declare the dependencies that our project relies on.

Lets first understand the BUILD LIFE CYCLE

Maven Build lifecycle phases:

- If you want to run "package" phase, all its previous phase will get executed first.
- And if you want to run specific goal of a particular phase, then all the goals of previous phases + current phase goals before the one you defined will get run.



Maven already has Validate phase defined and its goal, but if we want to add more goals or task, then we can use `<build>` element. And add the goal to specific phase.

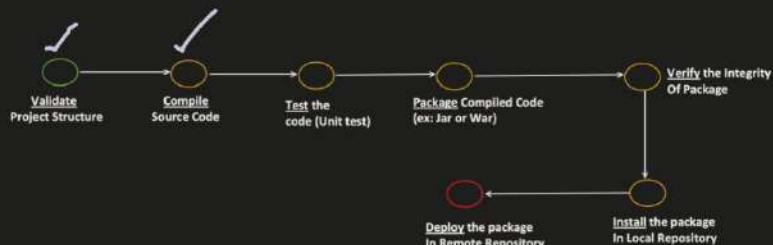
Validate:

**mvn validate**

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-checkstyle-plugin</artifactId>
  <version>3.1.2</version>
  <executions>
    <execution>
      <id>validate-checkstyle</id>
      <phase>validate</phase>
      <goals>
        <goal>check</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <configLocation>myCodeStyle.xml</configLocation>
  </configuration>
</plugin>
```

Compile:

**Run the command: mvn compile**



**It will validate and compile your code and put it under \${project.basedir}/target/classes**

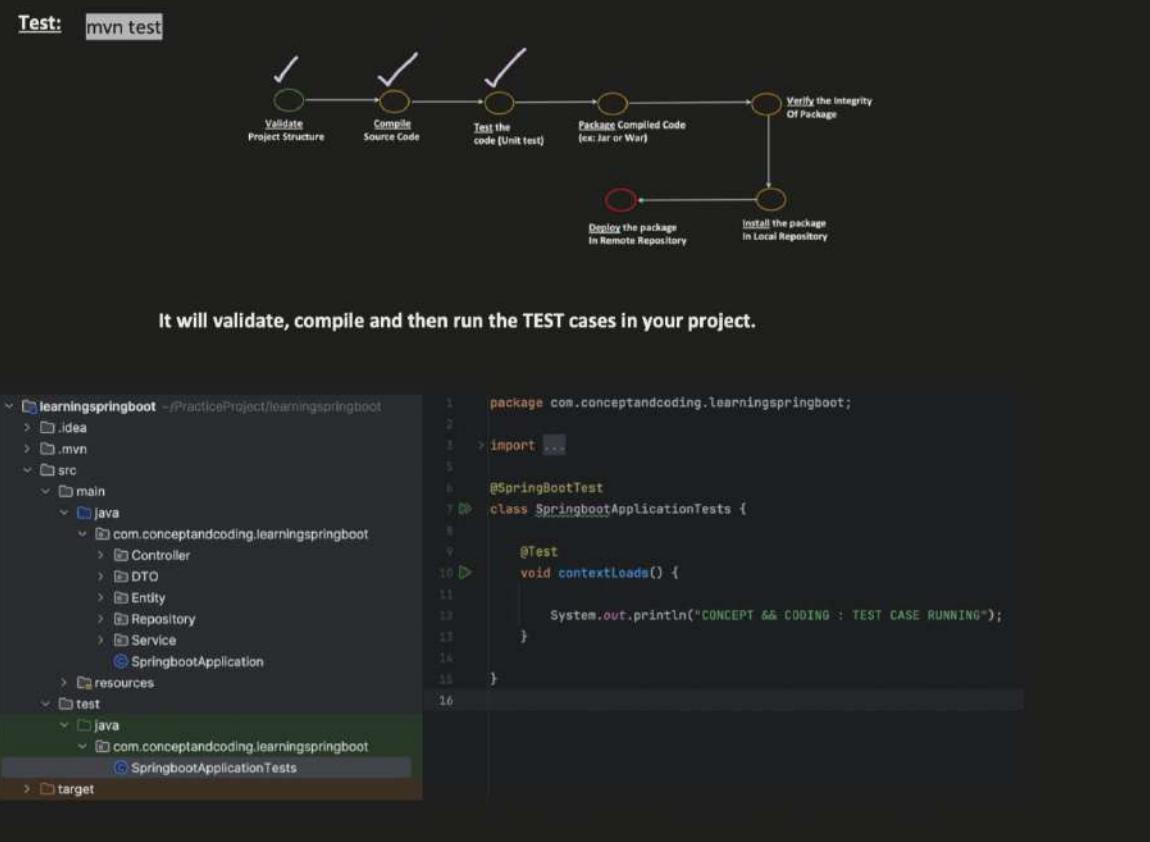
```
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.conceptandcoding:learningspringboot >-----
[INFO] Building springboot application 0.0.1-SNAPSHOT
[INFO]   from pom.xml
[INFO] ----- [ jar ] -----
[INFO]
[INFO] --- resources:3.3.1:resources (default-resources) @ learningspringboot ---
[INFO] Copying 1 resource from src/main/resources to target/classes
[INFO] Copying 0 resource from src/main/resources to target/classes
[INFO]
[INFO] --- compiler:3.11.0:compile (default-compile) @ learningspringboot ---
[INFO] Changes detected - recompiling the module! :source
[INFO] Compiling 7 source files with javac [debug release 17] to target/classes
[INFO]
[INFO] BUILD SUCCESS
[INFO]
[INFO] Total time: 0.812 s
[INFO] Finished at: 2024-03-09T16:35:24+05:30
[INFO] -----
```

After "mvn compile"

The image shows two side-by-side file explorers. The left one is labeled 'Before "mvn compile"' and the right one is labeled 'After "mvn compile"'. Both show a project structure for 'learningspringboot'. In the 'Before' state, the 'src/main/java' folder contains several packages like 'com.conceptandcoding.learningspringboot' with sub-packages 'Controller', 'DTO', 'Entity', 'Repository', 'Service', and 'SpringbootApplication'. A 'target/classes' folder is present but empty. In the 'After' state, the 'target/classes' folder is filled with numerous class files, indicating successful compilation of all source code.

Previously, "Ant" was popular, there we have to tell what to do and also how to do.

```
<project default="compile">
<target name="compile">
//some other properties here
<javac destdir="{provide your destination directory}">
<src>
<path element location="src/main/java"/>
</src>
//some other properties here
</javac>
</target>
</project>
```

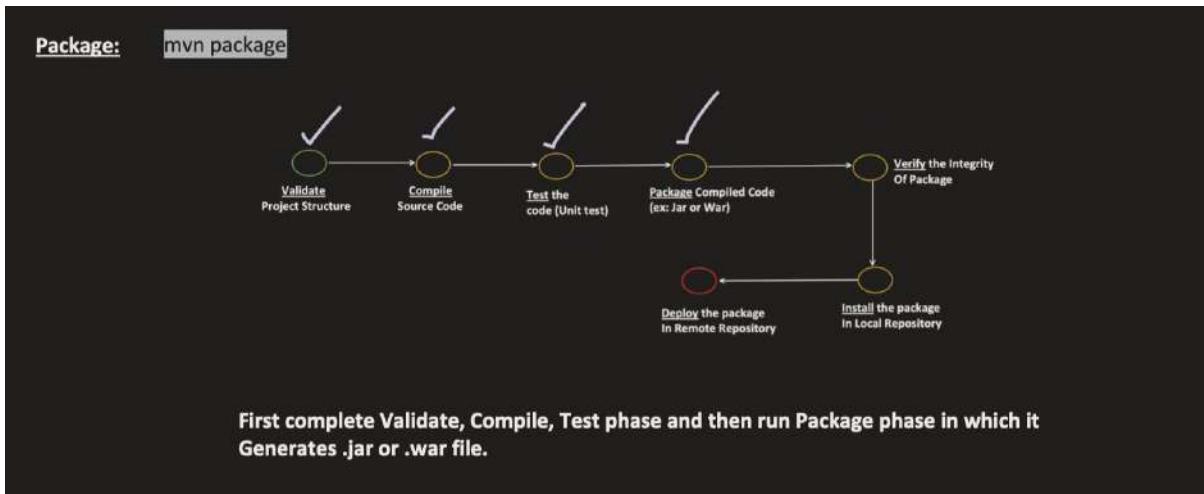


```

[INFO] Scanning for projects... 
[INFO]
[INFO] < com.conceptandcoding:learningspringboot >
[INFO] Building springboot application 0.0.1-SNAPSHOT
[INFO] from pom.xml
[INFO] ----- [ jar ] -----
[INFO]
[INFO] --- resources:3.3.1:resources (default-resources) @ learningspringboot ---
[INFO] Copying 1 resource from src/main/resources to target/classes
[INFO] Copying 0 resource from src/main/resources to target/classes
[INFO]
[INFO] --- compiler:3.11.0:compile (default-compile) @ learningspringboot ---
[INFO] Nothing to compile - all classes are up to date
[INFO] --- surefire:3.1.2:test (default-test) @ learningspringboot ---
[INFO] Using auto detected provider org.apache.maven.surefire.junitplatform.JUnitPlatformProvider
[INFO]
[INFO] -----
[INFO] T E S T S
[INFO] -----
[INFO] Running com.conceptandcoding.learningspringboot.SpringbootApplicationTests
2024-03-10T08:50:22.806+05:30 [main] INFO org.springframework.test.context.support.AnnotationConfigContextLoaderUtils -- tApplicationTests does not declare any static, non-private, non-final, nested classes annotated with @C
2024-03-10T08:50:22.655 [main] INFO org.springframework.boot.test.context.SpringBootTestTestContextBootstrapper -- Four
[INFO] :: Spring Boot ::          (v3.2.3)

2024-03-10T08:50:22.806+05:30  INFO 37695 --- [           main] c.c.l.SpringbootApplicationTests
gboot)
2024-03-10T08:50:22.806+05:30  INFO 37695 --- [           main] c.c.l.SpringbootApplicationTests
2024-03-10T08:50:23.292+05:30  INFO 37695 --- [           main] c.c.l.SpringbootApplicationTests
WARNING: If a serviceability tool is not in use, please run with -Djdk.instrument.traceUsage for more information
WARNING: Dynamic loading of agents will be disallowed by default in a future release
CONCEPT & CODING : TEST CASE RUNNING
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.235 s -- In com.conceptandcoding.learningspringboot.SpringbootApplicationTests
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time:  3.824 s
[INFO] Finished at: 2024-03-10T08:50:23+05:30
[INFO] -----

```



```

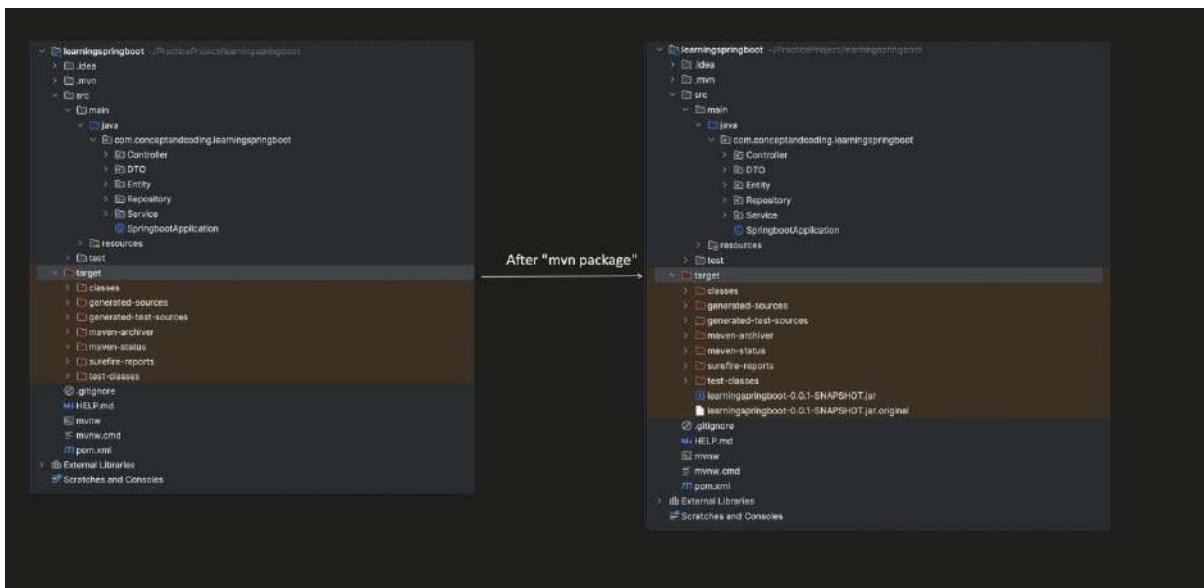
[INFO] Scanning for projects...
[INFO] -----
[INFO] < com.conceptandcoding:learningspringboot > -----
[INFO] Building springboot application 0.0.1-SNAPSHOT
[INFO]   from pom.xml
[INFO] -----
[INFO] --- resources:3.3.1:resources (default-resources) @ learningspringboot ---
[INFO] Copying 1 resource from src/main/resources to target/classes
[INFO] Copying 0 resource from src/main/resources to target/classes
[INFO]
[INFO] --- compiler:3.11.0:compile (default-compile) @ learningspringboot ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- resources:3.3.1:testResources (default-testResources) @ learningspringboot ---
[INFO] --- compiler:3.11.0:testCompile (default-testCompile) @ learningspringboot ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- surefire:3.1.2:test (default-test) @ learningspringboot ---
[INFO] Using auto detected provider org.apache.maven.surefire.junitplatform.JUnitPlatformProvider
[INFO]
[INFO] -----
[INFO] T E S T S
[INFO] -----



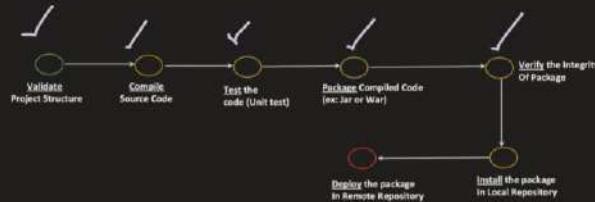
:: Spring Boot ::          (v3.2.3)

2024-03-10T09:05:27.378+05:30 INFO 38242 --- [           main] c.c.l.SpringbootApplicationTests
gboot)
2024-03-10T09:05:27.378+05:30 INFO 38242 --- [           main] c.c.l.SpringbootApplicationTests
2024-03-10T09:05:27.847+05:30 INFO 38242 --- [           main] c.c.l.SpringbootApplicationTests
OpenJDK 64-Bit Server VM warning: Sharing is only supported for boot loader classes because bootstrap
WARNING: If a serviceability tool is in use, please run with -XX:+EnableDynamicAgentLoading to hide this warning
WARNING: If a serviceability tool is not in use, please run with -Djdk.instrument.traceUsage for more information
WARNING: Dynamic loading of agents will be disallowed by default in a future release
CONCEPT & CODING : TEST CASE RUNNING
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.177 s -- in com.conceptandcoding.learningspringboot.
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
[INFO] --- jar:3.3.0:jar (default-jar) @ learningspringboot ---
[INFO] Building jar: /Users/PracticeProject/learningspringboot/target/learningspringboot-0.0.1-SNAPSHOT.jar

```



**Verify:** mvn verify



It can perform some additional checks apart from unit test cases like:

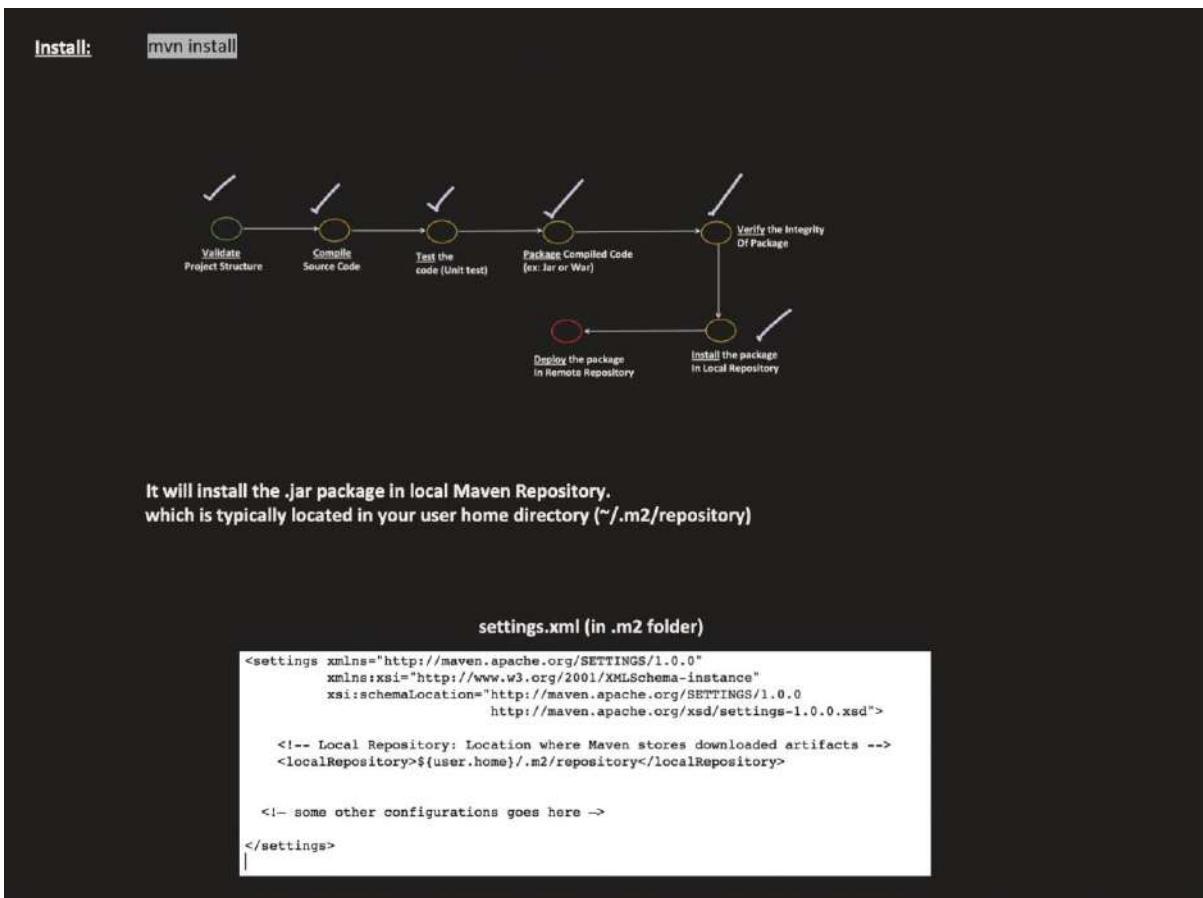
- STATIC CODE ANALYSIS
- CHECKSUM VERIFICATION etc...

#### STATIC CODE ANALYSIS:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-pmd-plugin</artifactId>
  <version>3.21.2</version>
  <executions>
    <execution>
      <id>pmd-analysis</id>
      <phase>verify</phase>
      <goals>
        <goal>pmd</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

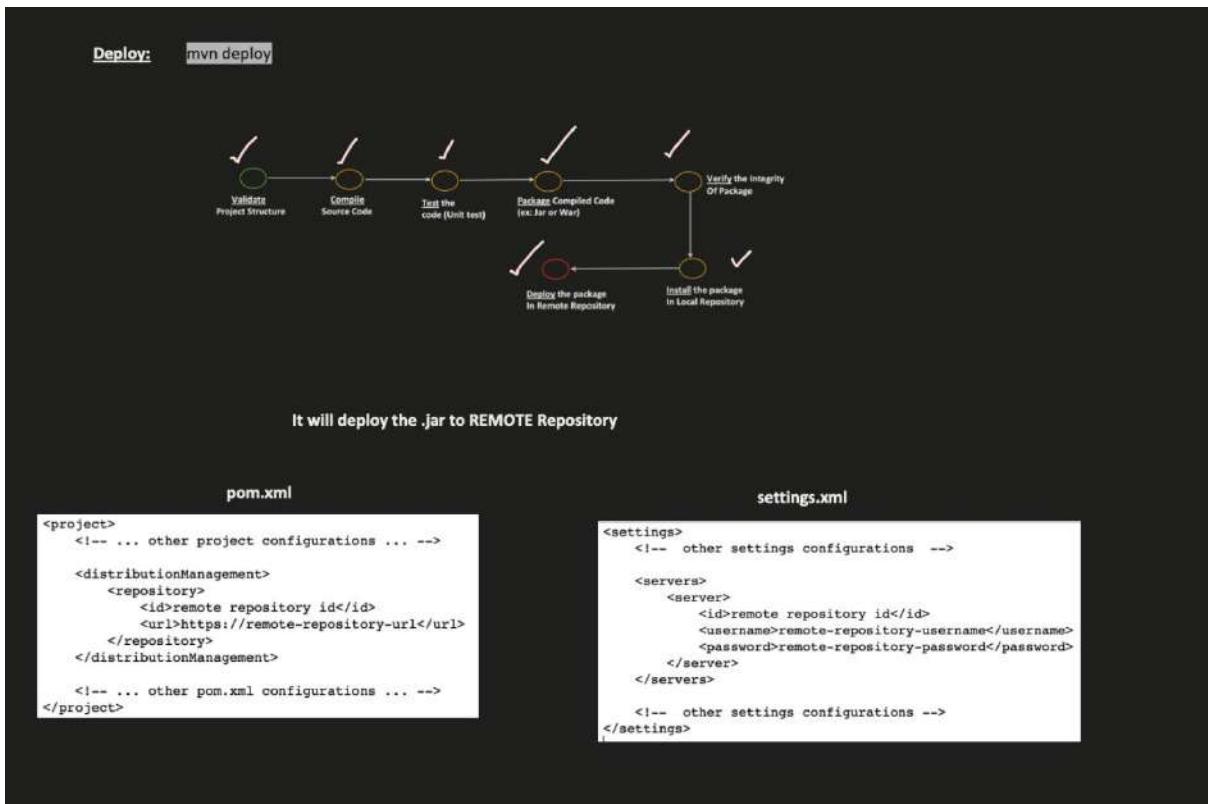
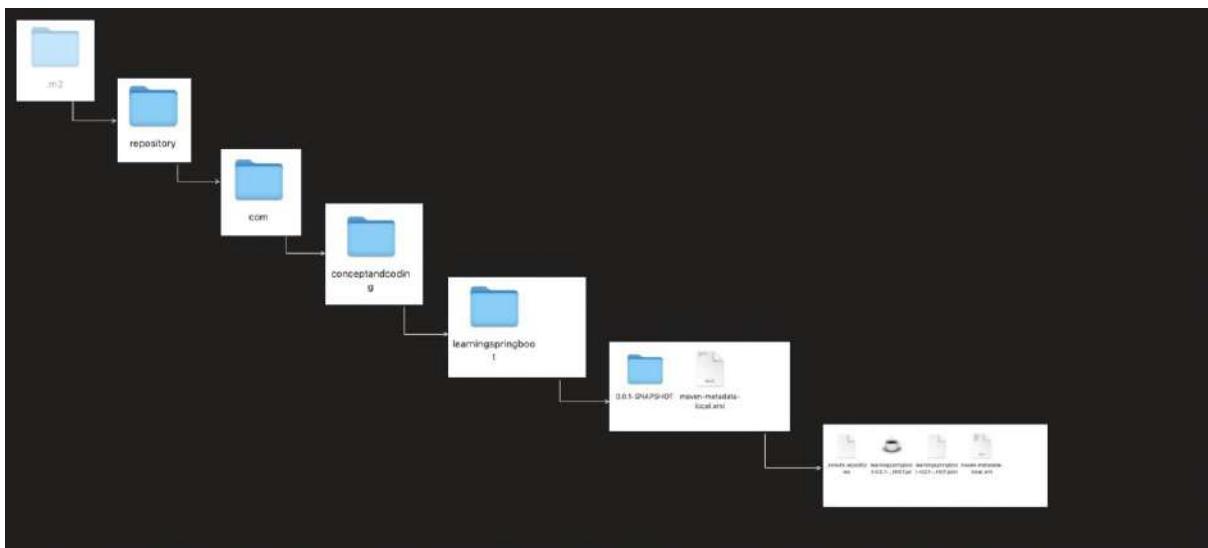
#### PMD is a source code analyzer:

- o Finds unused variable
- o Finds unused imports
- o Empty Catch block
- o No usage of object
- o Finds duplicate code etc...



```

[INFO] Scanning for projects...
[INFO]
[INFO] -----> com.conceptandcoding:learningspringboot <-----
[INFO] Building springboot application 0.0.1-SNAPSHOT
[INFO]   from pom.xml
[INFO] -----[ jar ]-----
[INFO] --- resources:3.3.1:resources (default-resources) @ learningspringboot ---
[INFO] Copying 1 resource from src/main/resources to target/classes
[INFO] Copying 0 resource from src/main/resources to target/classes
[INFO] --- compiler:3.11.0:compile (default-compile) @ learningspringboot ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- resources:3.3.1:testResources (default-testResources) @ learningspringboot ---
[INFO] --- compiler:3.11.0:testCompile (default-testCompile) @ learningspringboot ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- surefire:3.1.2:test (default-test) @ learningspringboot ---
[INFO] Using auto detected provider org.apache.maven.surefire.junitplatform.JUnitPlatformProvider
[INFO]
[INFO] ----- T E S T S -----
[INFO] -----[ Spring Boot ]----- (v3.2.3)
[INFO] 2024-03-10T09:05:27.378+05:30 [INFO] 38242 --- [           main] c.c.l.SpringbootApplicationTests
[INFO] 2024-03-10T09:05:27.378+05:30 [INFO] 38242 --- [           main] c.c.l.SpringbootApplicationTests
[INFO] 2024-03-10T09:05:27.642+05:30 [INFO] 38242 --- [           main] c.c.l.SpringbootApplicationTests
[INFO] OpenJDK 64-Bit Server VM warning: Sharing is only supported for boot loader classes because bootstrap class sharing cannot be disabled.
[INFO] WARNING: If a servicability tool is in use, please run with -XX:+DisableDynamicAgentLoading to hide this warning
[INFO] WARNING: Dynamic loading of agents will be disabled by default in a future release
[INFO] CONCEPT AA CODING : TEST CASE RUNNING
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.377 s --- in com.conceptandcoding.learningspringboot.
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] -----[ Jar ]----- learningspringboot
[INFO] 2024-03-10T09:05:27.642+05:30 [INFO] 38242 --- [           main] c.c.l.SpringbootApplicationTests
[INFO] 2024-03-10T09:05:27.642+05:30 [INFO] 38242 --- [           main] c.c.l.SpringbootApplicationTests
[INFO] Building jar: /Users/PracticeProject/Learningspringboot/target/learningspringboot-0.0.1-SNAPSHOT.jar
[INFO] --- install:3.1.1:install (default-install) @ learningspringboot ---
[INFO] --- install:3.1.1:install (default-install) @ learningspringboot ---
[INFO] --- install:3.1.1:install (default-install) @ learningspringboot ---
[INFO] Installing /Users/PracticeProject/Learningspringboot/target/learningspringboot-0.0.1-SNAPSHOT.jar to /Users/42/.m2/repository/com/conceptandcoding/learningspringboot/0.0.1-SNAPSHOT/learningspringboot-0.0.1-SNAPSHOT.jar
[INFO] BUILD SUCCESS
[INFO] Total time: 4.214 s
[INFO] Finished at: 2024-03-10T09:05:30
[INFO]
  
```



If we do not define the remote repository, then MAVEN during "mvn deploy" command we will face below error

```
[INFO] --- deploy:3.1.1:deploy (default-deploy) # learningSpringboot ---
[INFO] BUILD FAILURE
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  0.587 s
[INFO] Finalized at: 2024-03-16T13:00:11+01:30
[INFO]
[INFO] --- [Help 1]
[INFO] To see the full stack trace of the errors, re-run Maven with the -X switch.
[INFO] Re-run Maven using the -X switch to enable full debug logging.
[INFO]
[INFO] For more information about the errors and possible solutions, please read the following articles:
[INFO] [Help 2] http://cwiki.apache.org/confluence/display/MAVEN/MojoExecutionException
```

If we want to deploy the manifest to Maven central repository : <https://repo.maven.apache.org/maven2>. Since its public, we do not need username and password in settings.xml

## SpringBoot- Annotations (Controller Layer) Part1

1. **@Controller** : It indicates that the class is responsible for handling incoming HTTP requests.

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface Controller {
    @AliasFor(
        annotation = Component.class
    )
    String value() default "";
}
```

2. **@RestController** :

RestController = Controller + ResponseBody

3. **@ResponseBody**:

- Denotes that return value of the controller method should be serialized to HTTP response body.
- If we do not provide ResponseBody, Spring will consider response as name for the view and tries to resolve and render it (in case we are using the @Controller annotation)

4. **@RequestMapping**

- 1) Value, path (both are same)
- 2) Method
- 3) Consumes, produces
- 4) @Mapping
- 5) @Reflective({ControllerMappingReflectiveProcessor.class})

```
@RequestMapping { path = "/fetchUser", method = RequestMethod.GET, consumes = "application/json", produces = "application/json" }
```

```
@Target({ ElementType.TYPE, ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Mapping
@Reflective({ControllerMappingReflectiveProcessor.class})
public @interface RequestMapping {
    String name() default "";

    @AliasFor("path")
    String[] value() default {};

    @AliasFor("value")
    String[] path() default {};

    RequestMethod[] method() default {};

    String[] params() default {};

    String[] headers() default {};

    String[] consumes() default {};

    String[] produces() default {};
}
```

5. `@RequestParam`: Used to bind request parameter to controller method parameter.

<http://localhost:8080/api/fetchUser?firstName=SHRAYANSH&lastName=JAIN&age=32>

```
@RestController
@RequestMapping(value = "/api/")
public class SampleController {

    @GetMapping(path = "/fetchUser")
    public String getUserDetails(@RequestParam(name = "firstName") String firstName,
                                @RequestParam(name = "lastName", required = false) String lastName,
                                @RequestParam(name = "age") int age) {
        return "fetching and returning user details based on first name = " + firstName + ", lastName = " + lastName + " and age is = " + age;
    }
}
```

The framework automatically performs type conversion from the request parameter's string representation to the specified type.

1. Primitive types: Such as int, long, float, double, boolean, etc.
2. Wrapper classes: Such as Integer, Long, Float, Double, Boolean, etc.
3. String: Request parameters are inherently treated as strings only.
4. Enums: You can bind request parameters to enum types.
5. Custom object types: We can do it using a registered PropertyEditor.

#### How to used PropertyEditor?

```
@RestController
@RequestMapping(value = "/api/")
public class SampleController {

    @InitBinder
    protected void initBinder(DataBinder binder) {
        binder.registerCustomEditor(String.class, field: "firstName", new FirstNamePropertyEditor());
    }

    @GetMapping(path = "/fetchUser")
    public String getUserDetails(@RequestParam(name = "firstName") String firstName,
                                @RequestParam(name = "lastName", required = false) String lastName,
                                @RequestParam(name = "age") int age) {
        return "fetching and returning user details based on first name = " + firstName + ", lastName = " + lastName + " and age is = " + age;
    }
}
```

```
public class FirstNamePropertyEditor extends PropertyEditorSupport {
    @Override
    public void setAsText(String text) throws IllegalArgumentException {
        setValue(text.trim().toLowerCase());
    }
}
```

6. **@PathVariable:** Used to extract values from the path of the URL and help to bind it to controller method parameter.

```
@RestController
@RequestMapping(value = "/api/")
public class SampleController {

    @GetMapping(path = "/fetchUser/{firstName}")
    public String getUserDetails(@PathVariable(value = "firstName") String firstName) {
        return "fetching and returning user details based on first name = " + firstName;
    }
}
```

7. **@RequestBody:** Bind the body of HTTP request (typically JSON) to controller method parameter (java object).

```
@RestController
@RequestMapping(value = "/api/")
public class SampleController {

    @PostMapping(path = "/saveUser")
    public String getUserDetails(@RequestBody User user) {
        return "User created " + user.username + ":" + user.email;
    }
}
```

```
public class User {

    @JsonProperty ("user_name")
    String username;
    String email;

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
}
```

```
curl --location --request POST 'http://localhost:8080/api/saveUser' \
--header 'Content-Type: application/json' \
--data-raw '{
    "user_name": "Shrayansh",
    "email": "sjxyztest@gmail.com"
}'
```

8. ResponseEntity : It represents the entire HTTP response.

Header, status, response body etc.

```
@RestController
@RequestMapping(value = "/api/")
public class SampleController {

    @GetMapping(path = "/fetchUser")
    public ResponseEntity<String> getUserDetails(@RequestParam(value = "firstName") String firstName) {
        String output = "fetched User details of " + firstName;
        return ResponseEntity.status(HttpStatus.OK).body(output);
    }
}
```

## Spring boot : Bean and its Lifecycle

### What is bean?

In Simple term, Bean is a Java Object, which is managed by Spring container (also known as IOC Container).

IOC container -> contains all the beans which get created and also manage them.

### How to create a Bean?

@Component  
Annotation

@Bean  
Annotation

### 1. Using **@Component** Annotation:

- **@Component** annotation follows "convention over configuration" approach.
- Means Spring boot will try to auto configure based on conventions reducing the need for explicit configuration.
- **@Controller**, **@Service** etc. all are internally tells Spring to create bean and manage it.

So here, Spring boot will internally call **new User()** to create an instance of this class.

```
@Component
public class User {

    String username;
    String email;

    public String getUsername() { return username; }

    public void setUsername(String username) { this.username = username; }

    public String getEmail() { return email; }

    public void setEmail(String email) { this.email = email; }
}
```

## But what will happen to this now:

```
@Component
public class User {
    String username;
    String email;

    public User(String username, String email){
        this.username =username;
        this.email = email;
    }

    public String getUsername() { return username; }

    public void setUsername(String username) { this.username = username; }

    public String getEmail() { return email; }

    public void setEmail(String email) { this.email = email; }
}
```

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

Why? Because Spring boot does not know what to pass in these Constructor parameters.

So what to do?

@Bean comes into the picture, where we provide the configuration details and tells Spring boot to use it while creating a Bean.

```
public class User {  
    String username;  
    String email;  
  
    public User(String username, String email){  
        this.username =username;  
        this.email = email;  
    }  
  
    public String getUsername() { return username; }  
  
    public void setUsername(String username) { this.username = username; }  
  
    public String getEmail() { return email; }  
  
    public void setEmail(String email) { this.email = email; }  
}  
  
@Configuration  
public class AppConfig {  
  
    @Bean  
    public User createUserBean(){  
        return new User(username: "defaultusername", email: "defaultemail");  
    }  
}
```

If we add 2 times in Configuration file, Spring will create 2 beans of it.

```
@Configuration  
public class AppConfig {  
  
    @Bean  
    public User createUserBean(){  
        return new User(username: "defaultusername", email: "defaultemail");  
    }  
  
    @Bean  
    public User createAnotherUserBean(){  
        return new User(username: "anotherUsername", email: "anotheremail");  
    }  
}
```

Now, we know what is bean and how its getting created. But few Questions comes to our minds:

- > How Spring boot find these Beans?
- > At what time, these beans get created?

### **How Spring boot find these Beans?**

1. Using @ComponentScan annotation, it will scan the specified package and sub-package for classes annotated with @Component, @Service etc.

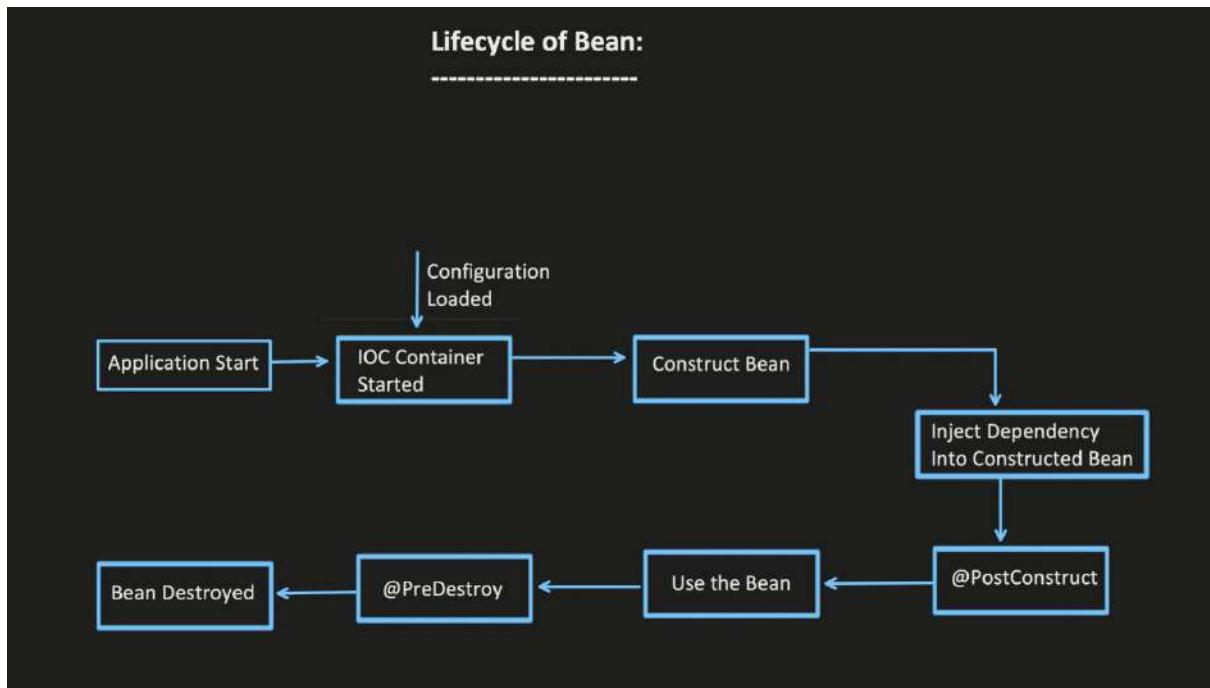
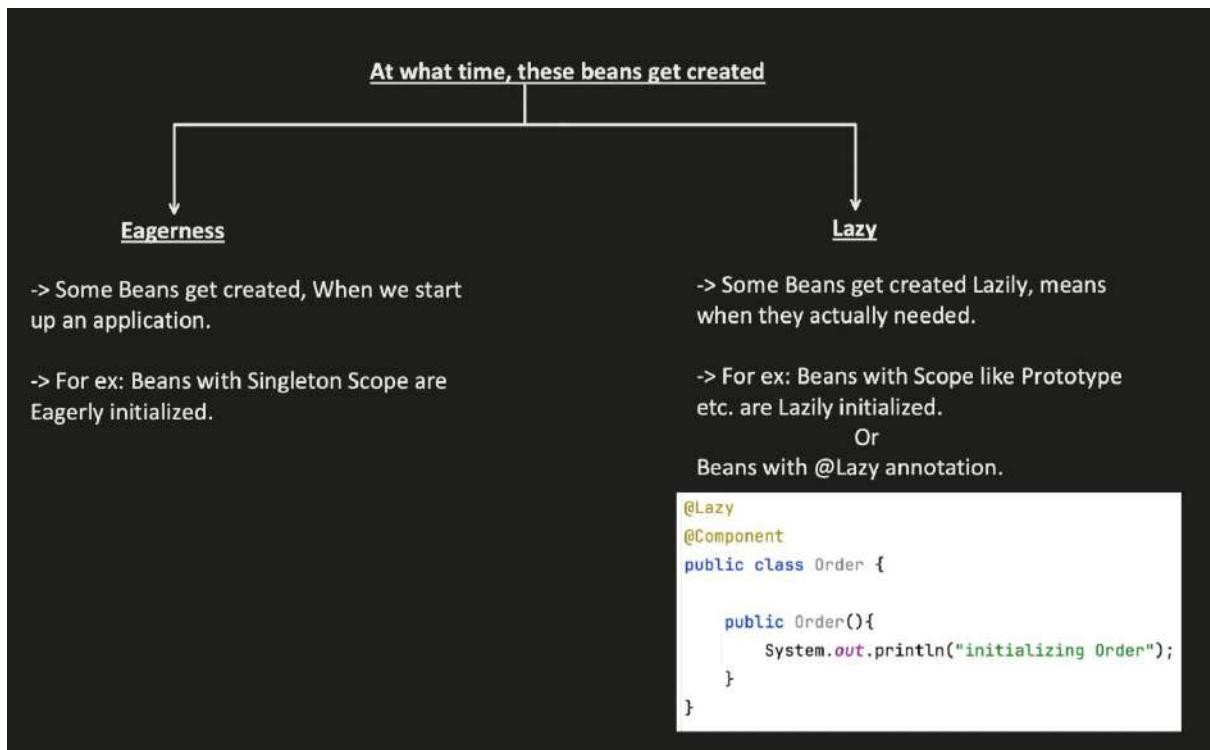
```
@SpringBootApplication
@ComponentScan(basePackages = "com.conceptandcoding.learningspringboot")
public class SpringbootApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringbootApplication.class, args);
    }
}
```

2. Through Explicit defining of bean via @Bean annotation in @Configuration class.

```
@Configuration
public class AppConfig {

    @Bean
    public User createUserBean(){
        return new User( username: "defaultusername", email: "defaultemail");
    }
}
```



### **Step1:**

- > During Application Startup, Spring boot invokes IOC Container  
*(ApplicationContext provides the implementation of IOC container)*
- > IOC Container, make use of Configuration and @ComponentScan to look out for the Classes for which beans need to be created.

*Invoking IOC Container*

```
main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port 8080 (http)
main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
main] o.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/10.1.19]
main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring Embedded WebApplicationContext
main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 419 ms
main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8080 (http) with context path ''
main] c.c.l.SpringbootApplication : Started SpringbootApplication in 0.771 seconds (process running for 0.946)
```

### **Step2: Construct the Beans**

```
@Component
public class User {

    public User(){
        System.out.println("initializing user");
    }
}
```

```
2024-04-09T00:47:43.681+05:30 INFO 60692 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port 8088 (http)
2024-04-09T00:47:43.687+05:30 INFO 60692 --- [           main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2024-04-09T00:47:43.687+05:30 INFO 60692 --- [           main] o.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/10.1.19]
2024-04-09T00:47:43.711+05:30 INFO 60692 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2024-04-09T00:47:43.711+05:30 INFO 60692 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 412 ms
initializing user
2024-04-09T00:47:43.859+05:30 INFO 60692 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8088 (http) with context path ''
2024-04-09T00:47:43.864+05:30 INFO 60692 --- [           main] c.c.l.SpringbootApplication : Started SpringbootApplication in 0.756 seconds (process running for 0.927)
```

### **Step3:**

- > Inject the Dependency into the Constructed Bean.
- > @Autowired, first look for a bean of the required type.
- > If bean found, Spring will inject it. Different ways of injection:
  - Constructor Injection
  - Setter Injection
  - Field Injection

\*\*\*\*\*We will see each injection and which one to use in next part \*\*\*\*\*

- > If bean is not found, Spring will create one and then inject it.

```
@Component  
public class User {  
  
    @Autowired  
    Order order;  
  
    public User(){  
        System.out.println("initializing user");  
    }  
}
```

```
@Lazy  
@Component  
public class Order {  
  
    public Order(){  
        System.out.println("Lazy: initializing Order");  
    }  
}
```

```
2024-04-09T00:53:11.751+05:30 INFO 68872 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port 8080 (http)  
2024-04-09T00:53:11.757+05:30 INFO 68872 --- [           main] o.apache.catalina.core.StandardService  : Starting service [Tomcat]  
2024-04-09T00:53:11.758+05:30 INFO 68872 --- [           main] o.apache.catalina.core.StandardEngine   : Starting Servlet engine: [Apache Tomcat/10.1.19]  
2024-04-09T00:53:11.782+05:30 INFO 68872 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext  
2024-04-09T00:53:11.782+05:30 INFO 68872 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 416 ms  
initializing user  
lazy: initializing Order  
2024-04-09T00:53:11.931+05:30 INFO 68872 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8080 (http) with context path ''  
2024-04-09T00:53:11.936+05:30 INFO 68872 --- [           main] c.c.l.SpringbootApplication            : Started SpringbootApplication in 0.76 seconds (process running for 0.929)  
!
```

**Step4:** Perform any task before Bean to be used in application.

```
@Component
public class User {

    @Autowired
    Order order;

    @PostConstruct
    public void initialize(){
        System.out.println("Bean has been constructed and dependencies have been injected");
    }

    public User(){
        System.out.println("initializing user");
    }
}
```

```
2024-04-09T01:01:14.147+05:30 INFO 61161 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port 8080 (http)
2024-04-09T01:01:14.152+05:30 INFO 61161 --- [           main] o.apache.catalina.core.StandardService  : Starting service [Tomcat]
2024-04-09T01:01:14.153+05:30 INFO 61161 --- [           main] o.apache.catalina.core.StandardEngine   : Starting Servlet engine: [Apache Tomcat/10.1.19]
2024-04-09T01:01:14.175+05:30 INFO 61161 --- [           main] o.a.c.C.[Tomcat].[localhost].[]       : Initializing Spring embedded WebApplicationContext
2024-04-09T01:01:14.176+05:30 INFO 61161 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 417 ms
initializing user
Lazy: initializing Order
Bean has been constructed and dependencies have been injected
2024-04-09T01:01:14.318+05:30 INFO 61161 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8080 (http) with context path ''
2024-04-09T01:01:14.322+05:30 INFO 61161 --- [           main] c.c.l.SpringbootApplication          : Started SpringbootApplication in 0.754 seconds (process running for 0.923)
```

**Step5:** Use the Bean in your application.

Means, we are using the bean (or Object) to invoke some methods to perform some business logic

**Step6:** Perform any task before Bean is getting destroyed.

```
@SpringBootApplication
public class SpringbootApplication {

    public static void main(String[] args) {
        ConfigurableApplicationContext context = SpringApplication.run(SpringbootApplication.class, args);
        context.close();
    }
}

@Component
public class User {

    @PostConstruct
    public void initialize(){ System.out.println("Post Construct initiated"); }

    @PreDestroy
    public void preDestroy(){ System.out.println("Bean is about to destroy, in PreDestroyMethod"); }

    public User(){ System.out.println("initializing user"); }
}
```

```
2024-04-09T18:07:37.600+05:30 INFO 16725 --- [           main] c.c.l.SpringbootApplication          : No active profile set, falling back to 1 default profile: "default"
2024-04-09T18:07:38.014+05:30 INFO 16725 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port 8080 (http)
2024-04-09T18:07:38.024+05:30 INFO 16725 --- [           main] o.apache.catalina.core.StandardService  : Starting service [Tomcat]
2024-04-09T18:07:38.024+05:30 INFO 16725 --- [           main] o.apache.catalina.core.StandardEngine   : Starting Servlet engine: [Apache Tomcat/10.1.19]
2024-04-09T18:07:38.048+05:30 INFO 16725 --- [           main] o.a.c.C.[Tomcat].[localhost].[]       : Initializing Spring embedded WebApplicationContext
2024-04-09T18:07:38.048+05:30 INFO 16725 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 426 ms
initializing user
Post Construct initiated
2024-04-09T18:07:38.197+05:30 INFO 16725 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8080 (http) with context path ''
2024-04-09T18:07:38.201+05:30 INFO 16725 --- [           main] c.c.l.SpringbootApplication          : Started SpringbootApplication in 0.764 seconds (process running for 0.931)
2024-04-09T18:07:49.215+05:30 WARN 16725 --- [           main] org.apache.tomcat.util.net.Acceptor   : The acceptor thread [http-nio-8080-Acceptor] did not stop cleanly
Bean is about to destroy, in PreDestroyMethod
```

# Springboot: Dependency Injection

## What is Dependency Injection

- Using Dependency injection, we can make our class independent of its dependencies.
- It helps to remove the dependency on concrete implementation and inject the dependencies from external source.

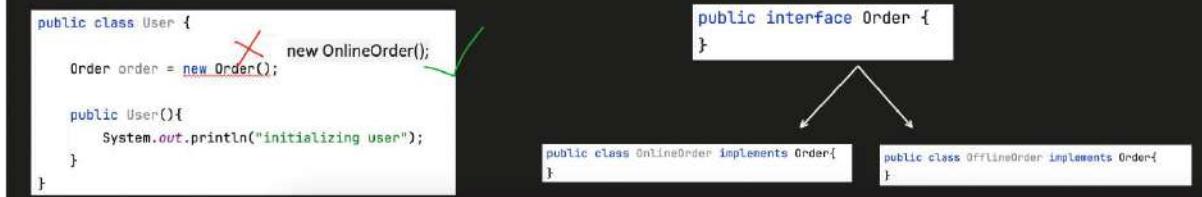
## Let's see the Problem first:

```
public class User {  
  
    Order order = new Order();  
  
    public User(){  
        System.out.println("initializing user");  
    }  
  
}  
  
public class Order {  
  
    public Order(){  
        System.out.println("initializing Order");  
    }  
  
}
```

### Issues with above class structure:

- Both User and Order class are **Tightly coupled**.

- Suppose, Order object creation logic gets changed (*lets say in future Object becomes an Interface and it has many concrete class*), then USER class has to be changed too.



- It breaks **Dependency Inversion Principle (DIP)**

- This principle says that DO NOT depend on concrete implementation, rather depends on abstraction.



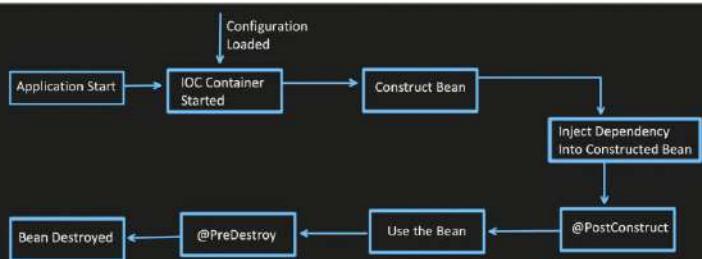
### Now in Spring boot how to achieve Dependency Inversion Principle?

#### Through **Dependency Injection**

- Using Dependency Injection, we can make our class independent of its dependencies.
- It helps to remove the dependency on concrete implementation and inject the dependencies from external source.

```
@Component  
public class User {  
  
    @Autowired  
    Order order;  
}  
  
@Component  
public class Order {  
}
```

`@Autowired`, first look for a bean of the required type.  
-> If bean found, Spring will inject it.



#### Different ways of Injection and which one is better?

- Field Injection
- Setter Injection
- Constructor Injection

#### Field Injection

- Dependency is set into the fields of the class directly.
- Spring uses reflection, it iterates over the fields and resolve the dependency.

```
@Component
public class User {

    @Autowired
    Order order;

    public User() {
        System.out.println("User initialized");
    }
}
```

```
@Component
@Lazy
public class Order {

    public Order(){
        System.out.println("order initialized");
    }
}
```

```
2024-04-13T21:33:30.390+05:30 INFO 20786 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port 8080 (http)
2024-04-13T21:33:30.397+05:30 INFO 20786 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2024-04-13T21:33:30.397+05:30 INFO 20786 --- [main] o.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/10.1.19]
2024-04-13T21:33:30.425+05:30 INFO 20786 --- [main] o.a.e.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2024-04-13T21:33:30.425+05:30 INFO 20786 --- [main] w.s.e.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 446 ms
User initialized
order initialized
2024-04-13T21:33:30.593+05:30 INFO 20786 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8080 (http) with context path ''
2024-04-13T21:33:30.598+05:30 INFO 20786 --- [main] c.c.l.SpringbootApplication : Started SpringbootApplication in 0.611 seconds (process running for 0.905)
```

#### Advantage:

- Very simple and easy to use.

#### Disadvantage:

- Can not be used with Immutable fields.

```
@Component
public class User {

    @Autowired
    public final Order order;

    public User() {
        System.out.println("User initialized");
    }
}
```

#### - Chances of NPE

```
@Component
public class User {

    @Autowired
    public Order order;

    public User() {
        System.out.println("User initialized");
    }

    public void process(){
        order.process();
    }
}
```

```
User userObj = new User();
userObj.process();
```

```
Exception in thread "main" java.lang.NullPointerException Create breakpoint :
at com.conceptandcoding.learningspringboot.User.process(User.java:18)
```

- During Unit Testing, setting MOCK dependency to this field becomes difficult.

```
@Component
public class User {
    @Autowired
    private Order order;

    public User() {
        System.out.println("User initialized");
    }

    public void process(){
        order.process();
    }
}
```

```
class UserTest {
    private Order orderMockObj;
    private User user;

    @BeforeEach
    public void setup(){
        this.orderMockObj = Mockito.mock(Order.class);
        this.user = new User();
    }
}
```

?? How to set this MOCK, we have to use reflection like  
@InjectMock annotation internally uses

```
class UserTest {
    @Mock
    private Order orderMockObj;

    @InjectMocks
    private User user;

    @BeforeEach
    public void setup(){
        MockitoAnnotations.initMocks(this);
    }
}
```

### Setter Injection

- Dependency is set into the fields using the setter method.
- We have to annotate the method using @Autowired

```
@Component
public class User {

    public Order order;

    public User() {
        System.out.println("User initialized");
    }

    @Autowired
    public void setOrderDependency(Order order){
        this.order = order;
    }
}
```

```
@Component
@Lazy
public class Order {

    public Order(){
        System.out.println("order initialized");
    }
}
```

#### Advantage:

- Dependency can be changed any time after the object creation (as object can not be marked as final).
- Ease of testing, as we can pass mock object in the dependency easily.

#### Disadvantage:

- Field Can not be marked as final. (We can not make it immutable).

```
@Component
public class User {

    public final Order order;

    public User() {
        System.out.println("User initialized");
    }

    @Autowired
    public void setOrderDependency(Order order){
        this.order = order;
    }
}
```

- Difficult to read and maintained, as per standard, object should be initialized during object creation, so this might create code readability issue.

#### **Constructor Injection**

- Dependency get resolved at the time of initialization of the Object itself.
- Its recommended to use

```

@Component
public class User {
    Order order;
    @Autowired
    public User(Order order) {
        this.order = order;
        System.out.println("User initialized");
    }
}

@Component
@Lazy
public class Order {
    public Order(){
        System.out.println("order initialized");
    }
}

2024-04-13T21:08:45.923+05:30 INFO 19992 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port 8080 (http)
2024-04-13T21:08:45.929+05:30 INFO 19992 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2024-04-13T21:08:45.929+05:30 INFO 19992 --- [main] o.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/10.1.19]
2024-04-13T21:08:45.955+05:30 INFO 19992 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2024-04-13T21:08:45.955+05:30 INFO 19992 --- [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 422 ms
order initialized
User initialized
2024-04-13T21:08:46.101+05:30 INFO 19992 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8080 (http) with context path ''
2024-04-13T21:08:46.105+05:30 INFO 19992 --- [main] c.c.l.SpringBootApplication : Started SpringbootApplication in 8.759 seconds (process running for 0.932)

```

When only 1 constructor is present, then using `@Autowired` on constructor is not mandatory. (from Spring version 4.3)

```

@Component
public class User {
    Order order;
    public User(Order order ) {
        this.order = order;
        System.out.println("User initialized");
    }
}

@Component
@Lazy
public class Order {
    public Order(){
        System.out.println("order initialized");
    }
}


```

```

2024-04-13T21:08:45.955+05:30 INFO 19992 --- [main] o.s.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2024-04-13T21:08:45.955+05:30 INFO 19992 --- [main] o.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 422 ms
order initialized
User initialized
2024-04-13T21:08:46.101+05:30 INFO 19992 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8080 (http) with context path ''
2024-04-13T21:08:46.105+05:30 INFO 19992 --- [main] c.c.l.SpringBootApplication : Started SpringbootApplication in 8.759 seconds (process running for 0.932)

```

When more than 1 constructor is present, then using `@Autowired` on constructor is mandatory.

```

@Component
public class User {
    Order order;
    Invoice invoice;
    public User(Order order ) {
        this.order = order;
        System.out.println("User initialized with only Order");
    }
    public User(Invoice invoice) {
        this.invoice = invoice;
        System.out.println("User initialized with only Invoice");
    }
}

@Component
@Lazy
public class Invoice {
    public Invoice() { System.out.println("invoice initialized"); }
}

@Component
@Lazy
public class Order {
    public Order(){
        System.out.println("order initialized");
    }
}

Caused by: org.springframework.beans.BeanInstantiationException: Failed to instantiate [com.conceptandcoding.learningspringboot.User]: No default constructor found

@Component
public class User {
    Order order;
    Invoice invoice;
    public User(Order order) {
        this.order = order;
        System.out.println("User initialized with only Order");
    }
    @Autowired
    public User(Invoice invoice) {
        this.invoice = invoice;
        System.out.println("User initialized with only Invoice");
    }
}

2024-04-13T21:16:01.654+05:30 INFO 20242 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2024-04-13T21:16:01.654+05:30 INFO 20242 --- [main] o.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/10.1.19]
2024-04-13T21:16:01.676+05:30 INFO 20242 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2024-04-13T21:16:01.677+05:30 INFO 20242 --- [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 430 ms
User initialized with only Order
User initialized with only Invoice
2024-04-13T21:16:01.827+05:30 INFO 20242 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8088 (http) with context path ''
2024-04-13T21:16:01.832+05:30 INFO 20242 --- [main] c.c.l.SpringBootApplication : Started SpringbootApplication in 8.775 seconds (process running for 0.94)

```

### Why Constructor Injection is Recommended (Advantages):

1. All mandatory dependencies are created at the time of initialization itself. Makes 100% sure that our object is fully initialized with mandatory dependency
  - i. avoid NPE during runtime
  - ii. Unnecessary null checks can be avoided too.
2. We can create immutable object using Constructor injection.

```
@Component
public class User { ✓
    public final Order order;

    @Autowired
    public User(Order order) {
        this.order = order;
        System.out.println("User initialized");
    }
}
```

```
@Component
public class User { ✗
    @Autowired
    public final Order order;

    public User() {
        System.out.println("User initialized");
    }
}
```

3. Fail Fast: If there is any missing dependency, it will fail during compilation itself, rather than failing during run Time.

```
@Component
public class User {
    public Order order;

    public User() {
        System.out.println("User initialized");
    }

    @PostConstruct
    public void init(){
        System.out.println(order == null);
    }
}
```

```
@Component
public class Order {
    public Order(){
        System.out.println("order initialized");
    }
}
```

**Using Constructor Injection, even if we missed @Autowired**

```
@Component
public class User {
    public Order order;

    public User(Order order) {
        this.order = order;
        System.out.println("User initialized");
    }

    @PostConstruct
    public void init(){
        System.out.println(order == null);
    }
}
```

OR

(it will fail fast, if Order bean is missing)

```
@Component
public class User {
    public Order order;

    public User(Order order) {
        this.order = order;
        System.out.println("User initialized");
    }

    @PostConstruct
    public void init(){
        System.out.println(order == null);
    }
}
```

```
public class Order {
    public Order(){
        System.out.println("order initialized");
    }
}
```

```
=====
APPLICATION FAILED TO START
=====
Description:
Parameter 0 of constructor in com.conceptualizing.learningspringboot.User requires a bean of type 'com.conceptualizing.learningspringboot.Order' that could not be found.
```

4. Unit testing is easy.

```
@Component
public class User {
    private Order order;

    @Autowired
    public User(Order order) {
        this.order = order;
        System.out.println("User initialized");
    }

    public void processOrder(){
        order.process();
    }
}
```

```
class UserTest {
    private Order orderMockObj;

    private User user;

    @BeforeEach
    public void setup(){
        this.orderMockObj = Mockito.mock(Order.class);
        this.user = new User(orderMockObj);
    }
}
```

## Common Issues when dealing with Dependency Injection:

### 1. CIRCULAR DEPENDENCY

```
@Component  
public class Order {  
  
    @Autowired  
    Invoice invoice;  
  
    public Order() {  
        System.out.println("order initialized");  
    }  
}
```

```
@Component  
public class Invoice {  
  
    @Autowired  
    Order order;  
  
    public Invoice() {  
        System.out.println("invoice initialized");  
    }  
}
```

```
*****  
APPLICATION FAILED TO START  
*****  
  
Description:  
  
The dependencies of some of the beans in the application context form a cycle:  
  
    [ ]  
    | invoice  
    ↑ ↓  
    | order  
    [ ]
```

## **Solutions:**

### **1. First and foremost, can we refactor the code and remove this cycle dependency:**

For example, common code in which both are dependent, can be taken out to separate class. This way we can break the circular dependency.

### **2. Using @Lazy on @Autowired annotation .**

Spring will create proxy bean instead of creating the bean instance immediately during application startup.

#### ***@Lazy on field Injection***

Let's first consider this

```
@Component
@Lazy
public class Order {

    public Order() {
        System.out.println("Order initialized");
    }
}

@Component
public class Invoice {

    @Autowired
    public Order order;

    public Invoice() {
        System.out.println("Invoice initialized");
    }
}
```

2024-04-14T18:21:55.967+05:30 INFO 48155 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2024-04-14T18:21:55.967+05:30 INFO 48155 --- [main] o.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/10.1.19]
2024-04-14T18:21:55.993+05:30 INFO 48155 --- [main] o.a.c.c.c.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2024-04-14T18:21:55.993+05:30 INFO 48155 --- [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 438 ns
Invoice initialized
Order initialized
2024-04-14T18:21:56.341+05:30 INFO 48155 --- [main] s.e.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8080 (http) with context path ''
2024-04-14T18:21:56.346+05:30 INFO 48155 --- [main] c.c.l.SpringBootTest : Started SpringbootApplication in 0.778 seconds (process running for 0.967)

Now, lets see this:

```
@Component
public class Invoice {

    @Lazy
    @Autowired
    public Order order;

    public Invoice() {
        System.out.println("Invoice initialized");
    }
}

@Component
@Lazy
public class Order {

    public Order() {
        System.out.println("Order initialized");
    }
}
```

2024-04-14T18:24:03.474+05:30 INFO 48250 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port 8080 (http)
2024-04-14T18:24:03.482+05:30 INFO 48250 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2024-04-14T18:24:03.482+05:30 INFO 48250 --- [main] o.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/10.1.19]
2024-04-14T18:24:03.507+05:30 INFO 48250 --- [main] o.a.c.c.c.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2024-04-14T18:24:03.507+05:30 INFO 48250 --- [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 426 ms
Invoice initialized
2024-04-14T18:24:03.677+05:30 INFO 48250 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8080 (http) with context path ''
2024-04-14T18:24:03.683+05:30 INFO 48250 --- [main] c.c.l.SpringBootTest : Started SpringbootApplication in 0.792 seconds (process running for 0.96)

Now, We can use this @Lazy to resolve the circular dependency

```
@Component
public class Order {
    @Autowired
    Invoice invoice;

    public Order() {
        System.out.println("order initialized");
    }
}

@Component
public class Invoice {
    @Lazy
    @Autowired
    public Order order;

    public Invoice() {
        System.out.println("Invoice initialized");
    }
}
```

```
2024-04-14T18:27:04.567+05:30 INFO 48425 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2024-04-14T18:27:04.568+05:30 INFO 48425 --- [main] o.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/10.1.19]
2024-04-14T18:27:04.591+05:30 INFO 48425 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2024-04-14T18:27:04.592+05:30 INFO 48425 --- [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 432 ms
Invoice initialized
Order initialized
2024-04-14T18:27:04.745+05:30 INFO 48425 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8080 (http) with context path ''
2024-04-14T18:27:04.758+05:30 INFO 48425 --- [main] c.c.l.SpringbootApplication : Started SpringbootApplication in 0.784 seconds (process running for 0.958)
```

### 3. Using @PostConstruct

```
@Component
public class Order {
    @Autowired
    Invoice invoice;

    public Order() {
        System.out.println("Order initialized");
    }

    @PostConstruct
    public void initialize(){
        invoice.setOrder(this);
    }
}

@Component
public class Invoice {
    public Order order;

    public Invoice() {
        System.out.println("Invoice initialized");
    }

    public void setOrder(Order order) {
        this.order = order;
    }
}
```

```
2024-04-14T18:27:04.567+05:30 INFO 48425 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2024-04-14T18:27:04.568+05:30 INFO 48425 --- [main] o.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/10.1.19]
2024-04-14T18:27:04.591+05:30 INFO 48425 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2024-04-14T18:27:04.592+05:30 INFO 48425 --- [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 432 ms
Invoice initialized
Order initialized
2024-04-14T18:27:04.745+05:30 INFO 48425 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8080 (http) with context path ''
2024-04-14T18:27:04.758+05:30 INFO 48425 --- [main] c.c.l.SpringbootApplication : Started SpringbootApplication in 0.784 seconds (process running for 0.958)
```

## UNSATISFIED DEPENDENCY

Problem:

```
@Component
public class User {
    @Autowired
    Order order;

    public User() {
        System.out.println("User initialized");
    }
}

public interface Order {
}

@Component
public class OnlineOrder implements Order {
    public OnlineOrder() {
        System.out.println("Online order initialized");
    }
}

@Component
public class OfflineOrder implements Order {
    public OfflineOrder() {
        System.out.println("Offline order initialized");
    }
}

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

UnsatisfiedDependencyException: Error creating bean with name 'user'

Solution:

### 1. @Primary annotation

```
@Component
public class User {
    @Autowired
    Order order;

    public User() {
        System.out.println("User initialized");
    }
}

@Primary
@Component
public class OnlineOrder implements Order {
    public OnlineOrder() {
        System.out.println("Online order initialized");
    }
}

@Component
public class OfflineOrder implements Order {
    public OfflineOrder() {
        System.out.println("Offline order initialized");
    }
}

2024-04-14T19:16:15.453+05:30 INFO 51489 --- [main] o.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/10.1.19]
2024-04-14T19:16:15.729+05:30 INFO 51489 --- [main] o.a.c.c.C.[Tomcat].[localhost] : Initializing Spring embedded WebApplicationContext
2024-04-14T19:16:15.730+05:30 INFO 51489 --- [main] o.a.c.c.C.[Tomcat].[localhost] : Root WebApplicationContext: initialization completed in 487 ms
Offline order initialized
User initialized
2024-04-14T19:16:15.874+05:30 INFO 51489 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port: 8080 (http) with context path: ''
2024-04-14T19:16:15.875+05:30 INFO 51489 --- [main] c.c.t.SpringbootApplication : Started SpringbootApplication in 9.765 seconds (process running for 8.932)
```

### 2. @Qualifier annotation

```
@Component
public class User {
    @Qualifier("offlineOrderName")
    @Autowired
    Order order;

    public User() {
        System.out.println("User initialized");
    }
}

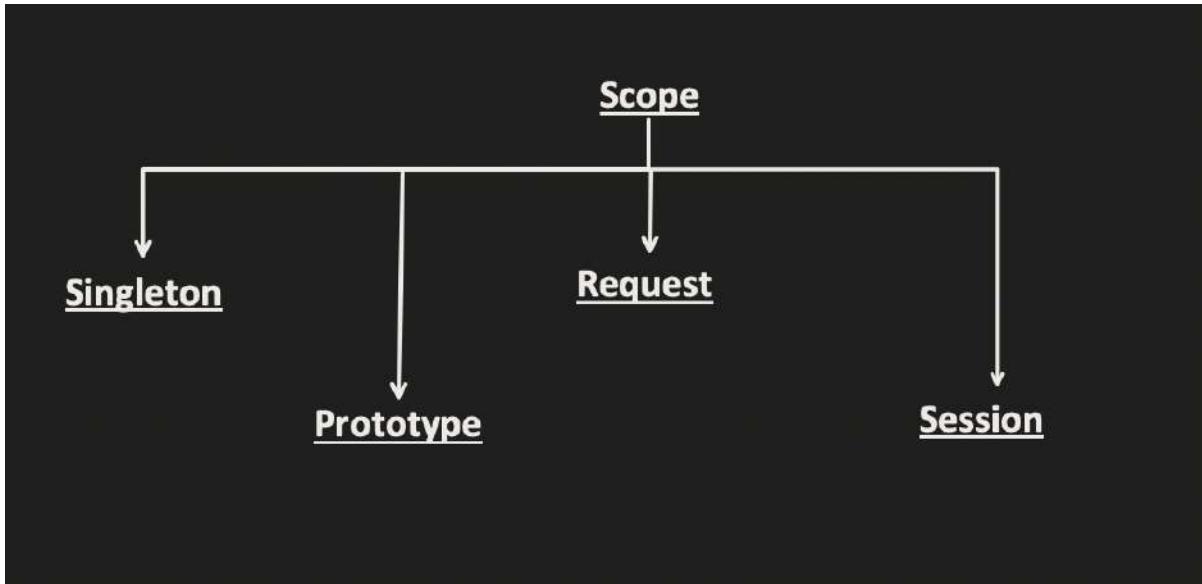
public interface Order {
}

@Component
@Qualifier("onlineOrderName")
public class OnlineOrder implements Order {
    public OnlineOrder() {
        System.out.println("Online order initialized");
    }
}

@Component
@Qualifier("offlineOrderName")
public class OfflineOrder implements Order {
    public OfflineOrder() {
        System.out.println("Offline order initialized");
    }
}

2024-04-14T19:16:15.453+05:30 INFO 51489 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2024-04-14T19:16:15.453+05:30 INFO 51489 --- [main] o.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/10.1.19]
2024-04-14T19:16:15.457+05:30 INFO 51489 --- [main] o.a.c.c.C.[Tomcat].[localhost] : Initializing Spring embedded WebApplicationContext
2024-04-14T19:16:15.457+05:30 INFO 51489 --- [main] o.a.c.c.C.[Tomcat].[localhost] : Root WebApplicationContext: initialization completed in 619 ms
Offline order initialized
User initialized
2024-04-14T19:16:15.807+05:30 INFO 51489 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port: 8080 (http) with context path: ''
2024-04-14T19:16:15.813+05:30 INFO 51489 --- [main] c.c.t.SpringbootApplication : Started SpringbootApplication in 9.765 seconds (process running for 8.932)
```

## Springboot: Bean Scopes by Concept & Coding



### Singleton:

- Default scope
- Only 1 instance created per IOC.
- Eagerly initialized by IOC (means at the time of application startup, object get created)

```


@Singleton
@ScopeDefinition(value = "singleton")
public class TestController2 {
    @Autowired
    User user;

    public TestController1() {
        System.out.println("TestController1 instance initialization");
    }

    @PostConstruct
    public void init() {
        System.out.println("TestController1 object hashCode: " + this.hashCode());
        System.out.println("User object hashCode: " + user.hashCode());
    }

    @RequestMapping(path = "/testController")
    public ResponseEntity<String> testController() {
        System.out.println("method name is invoked");
        return ResponseEntity.status(HttpStatus.OK).body("");
    }
}


```

```


@Singleton
@ScopeDefinition(value = "singleton")
@ScopeDefinitionFactory(SCOPE_SINGLETON)
public class TestController2 {
    @Autowired
    User user;

    public TestController1() {
        System.out.println("TestController2 instance initialization");
    }

    @PostConstruct
    public void init() {
        System.out.println("TestController2 object hashCode: " + this.hashCode());
        System.out.println("User object hashCode: " + user.hashCode());
    }

    @RequestMapping(path = "/testController")
    public ResponseEntity<String> testController() {
        System.out.println("method name is invoked");
        return ResponseEntity.status(HttpStatus.OK).body("");
    }
}


```

```


@Component
@Scope("singleton")
public class User {

    public User() {
        System.out.println("User initialization");
    }

    @PostConstruct
    public void init() {
        System.out.println("User object hashCode: " + this.hashCode());
    }

    @RequestMapping(path = "/username")
    public ResponseEntity<String> getUsername() {
        System.out.println("username is invoked");
        return ResponseEntity.status(HttpStatus.OK).body("User");
    }
}


```

```


2024-05-01T13:12:47.754+05:30 INFO 36566 --- [           main] c.c.l.SpringbootApplication
2024-05-01T13:12:47.754+05:30 INFO 36566 --- [           main] c.c.l.SpringbootApplication
2024-05-01T13:12:48.182+05:30 INFO 36566 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer
2024-05-01T13:12:48.188+05:30 INFO 36566 --- [           main] o.apache.catalina.core.StandardService
2024-05-01T13:12:48.188+05:30 INFO 36566 --- [           main] o.apache.catalina.core.StandardEngine
2024-05-01T13:12:48.215+05:30 INFO 36566 --- [           main] o.a.c.c.Tomcat.[localhost].[/]
2024-05-01T13:12:48.215+05:30 INFO 36566 --- [           main] w.s.c.ServletWebServerApplicationContext
TestController1 instance initialization
User initialization
User object hashCode: 1140202235
TestController1 object hashCode: 1046302571 User object hashCode: 1140202235
TestController2 instance initialization
TestController2 object hashCode: 1525241607 User object hashCode: 1140202235
2024-05-01T13:12:48.365+05:30 INFO 36566 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer
2024-05-01T13:12:48.369+05:30 INFO 36566 --- [           main] c.c.l.SpringbootApplication
2024-05-01T13:13:50.207+05:30 INFO 36566 --- [nio-8090-exec-1] o.a.c.c.Tomcat.[localhost].[/]
2024-05-01T13:13:50.207+05:30 INFO 36566 --- [nio-8090-exec-1] o.s.web.servlet.DispatcherServlet
2024-05-01T13:13:50.208+05:30 INFO 36566 --- [nio-8090-exec-1] o.s.web.servlet.DispatcherServlet


```

① localhost:8090/api/fetchUser

```

2024-05-01T13:12:47.754+05:30 INFO 36566 --- [           main] c.c.l.SpringbootApplication
2024-05-01T13:12:47.755+05:30 INFO 36566 --- [           main] c.c.l.SpringbootApplication
2024-05-01T13:12:48.182+05:30 INFO 36566 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer
2024-05-01T13:12:48.188+05:30 INFO 36566 --- [           main] o.apache.catalina.core.StandardService
2024-05-01T13:12:48.188+05:30 INFO 36566 --- [           main] o.apache.catalina.core.StandardEngine
2024-05-01T13:12:48.213+05:30 INFO 36566 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]
2024-05-01T13:12:48.213+05:30 INFO 36566 --- [           main] w.s.c.ServletWebServerApplicationContext
TestController1 instance initialization
User initialization
User object hashCode: 1140202235
TestController1 object hashCode: 1046302571 User object hashCode: 1140202235
TestController2 instance initialization
TestController2 object hashCode: 1525241607 User object hashCode: 1140202235
2024-05-01T13:12:48.365+05:30 INFO 36566 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer
2024-05-01T13:12:48.369+05:30 INFO 36566 --- [           main] c.c.l.SpringbootApplication
2024-05-01T13:13:50.207+05:30 INFO 36566 --- [nio-8090-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]
2024-05-01T13:13:50.207+05:30 INFO 36566 --- [nio-8090-exec-1] o.s.web.servlet.DispatcherServlet
2024-05-01T13:13:50.208+05:30 INFO 36566 --- [nio-8090-exec-1] o.s.web.servlet.DispatcherServlet
fetchUser api invoked

```

#### Prototype:

- Each time new Object is created.
- Its Lazily Initialized, means when object is created only when its required.

```

@PostConstruct
public void init() {
    System.out.println("Student object hashCode: " + this.hashCode() +
                       " User object hashCode: " + user.hashCode());
}
}

@Component
@Scope("prototype")
public class User {
    ...
}

@Component
public class Student {
    ...
    @Autowired
    User user;

    public Student() {
        System.out.println("Student instance initialization");
    }

    @PostConstruct
    public void init() {
        System.out.println("Student object hashCode: " + this.hashCode() +
                           " User object hashCode: " + user.hashCode());
    }
}

```

```

@PostConstruct
public void init() {
    System.out.println("TestController1 instance initialization");
}

@PostConstruct
public void init() {
    System.out.println("TestController1 object hashCode: " + this.hashCode() +
                       " User object hashCode: " + user.hashCode() +
                       " Student object hashCode: " + student.hashCode());
}

@GetMapping(path = "/fetchUser")
public ResponseEntity<String> getUserDetails() {
    System.out.println("fetchUser api invoked");
    return ResponseEntity.status(HttpStatus.OK).body("");
}
}

```

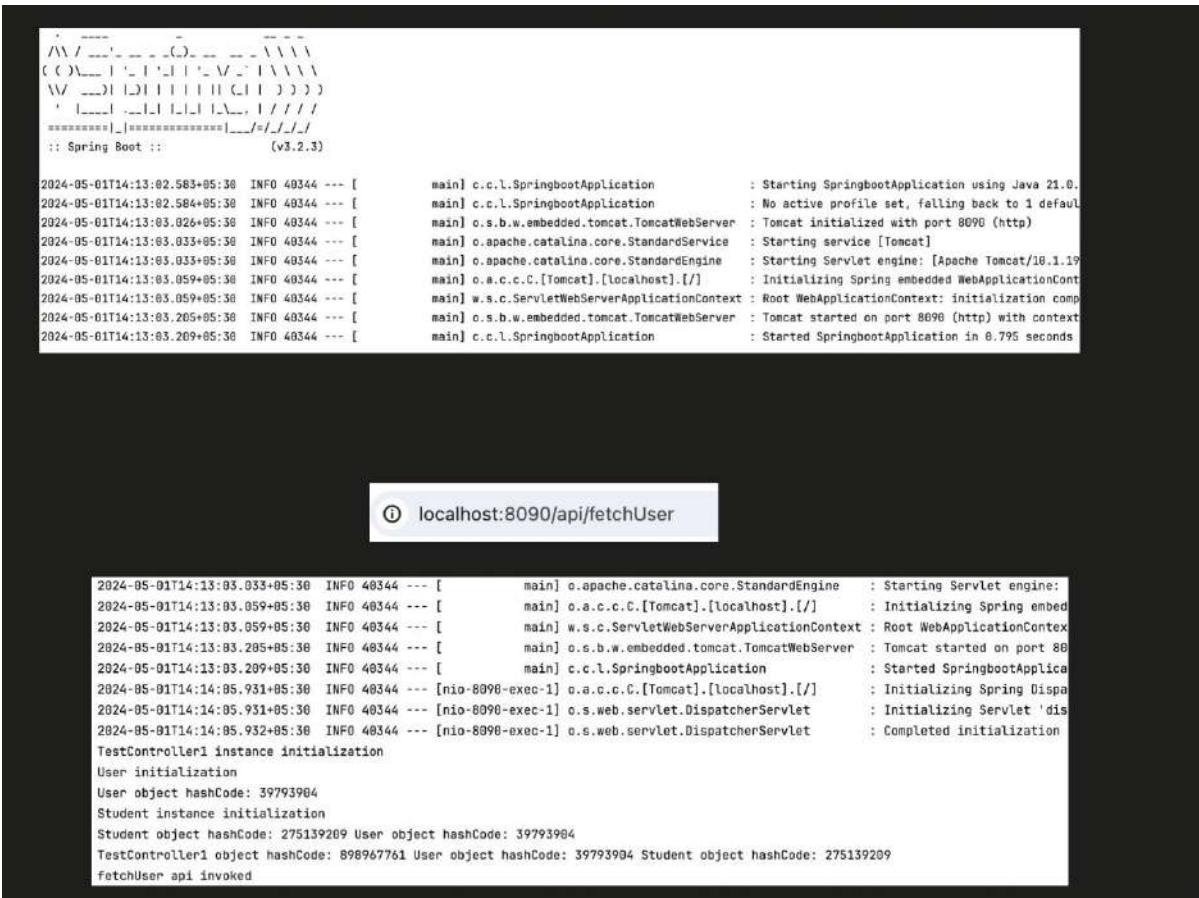
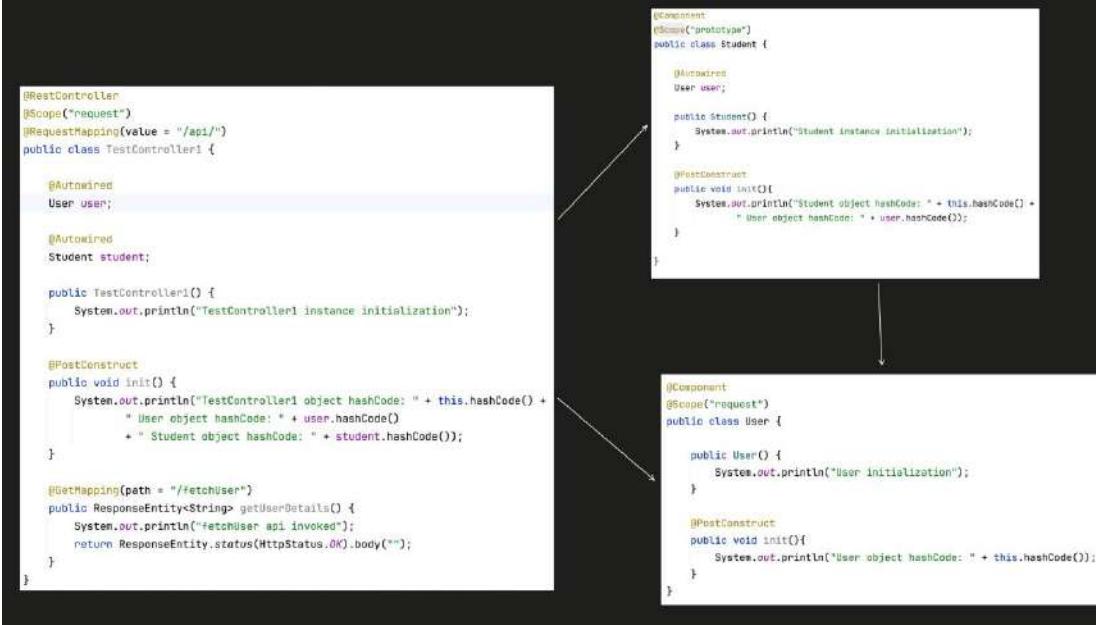
```
2024-05-01T13:32:00.689+05:30 INFO 37722 --- [           main] c.c.l.SpringbootApplication
2024-05-01T13:32:00.691+05:30 INFO 37722 --- [           main] c.c.l.SpringbootApplication
2024-05-01T13:32:01.130+05:30 INFO 37722 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer
2024-05-01T13:32:01.136+05:30 INFO 37722 --- [           main] o.apache.catalina.core.StandardService
2024-05-01T13:32:01.136+05:30 INFO 37722 --- [           main] o.apache.catalina.core.StandardEngine
2024-05-01T13:32:01.160+05:30 INFO 37722 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]
2024-05-01T13:32:01.160+05:30 INFO 37722 --- [           main] w.s.c.ServletWebServerApplicationContext
Student instance initialization
User initialization
User object hashCode: 1510009630
Student object hashCode: 2092450685 User object hashCode: 1510009630
2024-05-01T13:32:01.328+05:30 INFO 37722 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer
2024-05-01T13:32:01.333+05:30 INFO 37722 --- [           main] c.c.l.SpringbootApplication
```

#### ① localhost:8090/api/fetchUser

```
2024-05-01T13:32:00.689+05:30 INFO 37722 --- [           main] c.c.l.SpringbootApplication : Starting
2024-05-01T13:32:00.691+05:30 INFO 37722 --- [           main] c.c.l.SpringbootApplication : No activ
2024-05-01T13:32:01.130+05:30 INFO 37722 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat i
2024-05-01T13:32:01.136+05:30 INFO 37722 --- [           main] o.apache.catalina.core.StandardService : Starting
2024-05-01T13:32:01.136+05:30 INFO 37722 --- [           main] o.apache.catalina.core.StandardEngine : Starting
2024-05-01T13:32:01.160+05:30 INFO 37722 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initiali
2024-05-01T13:32:01.160+05:30 INFO 37722 --- [           main] w.s.c.ServletWebServerApplicationContext : Root Web
Student instance initialization
User initialization
User object hashCode: 1510009630
Student object hashCode: 2092450685 User object hashCode: 1510009630
2024-05-01T13:32:01.328+05:30 INFO 37722 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat s
2024-05-01T13:32:01.333+05:30 INFO 37722 --- [           main] c.c.l.SpringbootApplication : Started
2024-05-01T13:32:50.142+05:30 INFO 37722 --- [nio-8090-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initiali
2024-05-01T13:32:50.142+05:30 INFO 37722 --- [nio-8090-exec-1] o.s.web.servlet.DispatcherServlet : Initiali
2024-05-01T13:32:50.143+05:30 INFO 37722 --- [nio-8090-exec-1] o.s.web.servlet.DispatcherServlet : Complete
TestController1 instance initialization
User initialization
User object hashCode: 1984730322
TestController1 object hashCode: 1786739287 User object hashCode: 1984730322 Student object hashCode: 2092450685
fetchUser api invoked
```

**Request:**

- New Object is created for each HTTP request.
- Lazily initialized.

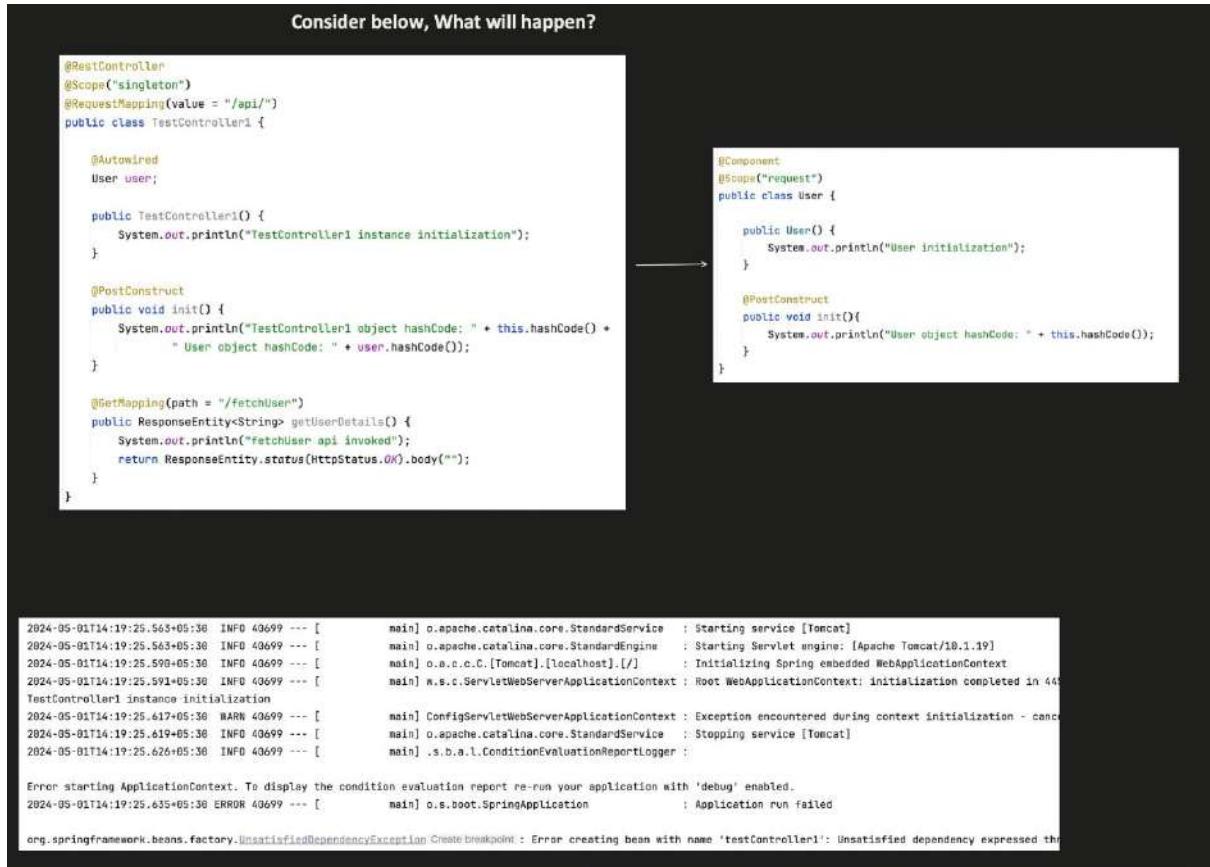


① localhost:8090/api/fetchUser

```

2024-05-01T14:13:03.205+05:30 INFO 40344 --- [main] o.s.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8090 (http) with context path ''
2024-05-01T14:13:03.209+05:30 INFO 40344 --- [main] c.c.l.SpringbootApplication : Started SpringbootApplication in 0.011 seconds (JVM: 0.011s)
2024-05-01T14:14:05.931+05:30 INFO 40344 --- [nio-8090-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring FrameworkServlet 'dispatcherServlet'
2024-05-01T14:14:05.931+05:30 INFO 40344 --- [nio-8090-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
2024-05-01T14:14:05.932+05:30 INFO 40344 --- [nio-8090-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 1ms
TestController1 instance initialization
User initialization
User object hashCode: 39793904
Student instance initialization
Student object hashCode: 275139209 User object hashCode: 39793904
TestController1 object hashCode: 898967761 User object hashCode: 39793904 Student object hashCode: 275139209
fetchUser api invoked
TestController1 instance initialization
User initialization
User object hashCode: 1227388929
Student instance initialization
Student object hashCode: 1206886228 User object hashCode: 1227388929
TestController1 object hashCode: 1137709937 User object hashCode: 1227388929 Student object hashCode: 1206886228
fetchUser api invoked

```



Spring creates Request scope bean only when there is active HTTP request present.  
Since Singleton, is eagerly initialized, there is no active HTTP request present in  
current thread. So it wont create a bean for User.

```
@RestController
@Scope("singleton")
@RequestMapping(value = "/api/")
public class TestController1 {

    @Autowired
    User user;

    public TestController1() {
        System.out.println("TestController1 instance initialization");
    }

    @PostConstruct
    public void init() {
        System.out.println("TestController1 object hashCode: " + this.hashCode() +
                           " User object hashCode: " + user.hashCode());
    }

    @GetMapping(path = "/fetchUser")
    public ResponseEntity<String> getUserDetails() {
        System.out.println("fetchUser api invoked");
        return ResponseEntity.status(HttpStatus.OK).body("");
    }
}
```

```
@Component
@RequestMapping(value = "request", proxyMode = ScopedProxyMode.TARGET_CLASS)
public class User {

    public User() {
        System.out.println("User initialization");
    }

    @PostConstruct
    public void init(){
        System.out.println("User object hashCode: " + this.hashCode());
    }

    public void dummyMethod(){
    }
}
```

```
.....
:: Spring Boot ::      (v3.2.3)

2024-05-01T15:16:24.864+05:30  INFO 43839 --- [           main] c.c.l.SpringbootApplication               : main] c.c.l.SpringbootApplication
2024-05-01T15:16:24.865+05:30  INFO 43839 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : main] o.s.b.w.embedded.tomcat.TomcatWebServer
2024-05-01T15:16:25.318+05:30  INFO 43839 --- [           main] o.apache.catalina.core.StandardService   : main] o.apache.catalina.core.StandardService
2024-05-01T15:16:25.325+05:30  INFO 43839 --- [           main] o.apache.catalina.core.StandardEngine     : main] o.apache.catalina.core.StandardEngine
2024-05-01T15:16:25.351+05:30  INFO 43839 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : main] o.a.c.c.C.[Tomcat].[localhost].[/]
2024-05-01T15:16:25.352+05:30  INFO 43839 --- [           main] w.s.c.ServletWebServerApplicationContext  : main] w.s.c.ServletWebServerApplicationContext
TestController1 instance initialization
TestController1 object hashCode: 1356419559 User object hashCode: 1159352444
2024-05-01T15:16:25.534+05:30  INFO 43839 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : main] o.s.b.w.embedded.tomcat.TomcatWebServer
2024-05-01T15:16:25.539+05:30  INFO 43839 --- [           main] c.c.l.SpringbootApplication               : main] c.c.l.SpringbootApplication
```

① localhost:8090/api/fetchUser

```
2024-05-01T15:16:24.864+05:30 INFO 43839 --- [           main] c.c.l.SpringbootApplication
2024-05-01T15:16:24.865+05:30 INFO 43839 --- [           main] c.c.l.SpringbootApplication
2024-05-01T15:16:25.318+05:30 INFO 43839 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer
2024-05-01T15:16:25.325+05:30 INFO 43839 --- [           main] o.apache.catalina.core.StandardService
2024-05-01T15:16:25.325+05:30 INFO 43839 --- [           main] o.apache.catalina.core.StandardEngine
2024-05-01T15:16:25.351+05:30 INFO 43839 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]
2024-05-01T15:16:25.352+05:30 INFO 43839 --- [           main] w.s.c.ServletWebServerApplicationContext
TestController1 instance initialization
TestController1 object hashCode: 1356419559 User object hashCode: 1159352444
2024-05-01T15:16:25.534+05:30 INFO 43839 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer
2024-05-01T15:16:25.539+05:30 INFO 43839 --- [           main] c.c.l.SpringbootApplication
2024-05-01T15:16:50.198+05:30 INFO 43839 --- [nio-8090-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]
2024-05-01T15:16:50.198+05:30 INFO 43839 --- [nio-8090-exec-1] o.s.web.servlet.DispatcherServlet
2024-05-01T15:16:50.199+05:30 INFO 43839 --- [nio-8090-exec-1] o.s.web.servlet.DispatcherServlet
fetchUser api invoked
User initialization
User object hashCode: 1078757370
```

#### Session:

- New Object is created for each HTTP session.
- Lazily initialized.
- When user accesses any endpoint, session is created.
- Remains active, till it does not expires.

```
@RestController
@Scope(value = "session")
@RequestMapping(value = "/api/")
public class TestController1 {

    @Autowired
    User user;

    public TestController1() {
        System.out.println("TestController1 instance initialization");
    }

    @PostConstruct
    public void init() {
        System.out.println("TestController1 object hashCode: " + this.hashCode() +
                           " User object hashCode: " + user.hashCode());
    }

    @GetMapping(path = "/fetchUser")
    public ResponseEntity<String> getUserDetails() {
        System.out.println("fetchUser api invoked");
        return ResponseEntity.status(HttpStatus.OK).body("");
    }

    @GetMapping(path = "/logout")
    public ResponseEntity<String> getUserDetails(HttpServletRequest request) {
        System.out.println("end the session");
        HttpSession session = request.getSession();
        session.invalidate();
        return ResponseEntity.status(HttpStatus.OK).body("");
    }
}
```

```
@Component
public class User {

    public User() {
        System.out.println("User initialization");
    }

    @PostConstruct
    public void init(){
        System.out.println("User object hashCode: " + this.hashCode());
    }

    public void dummyMethod(){
    }
}
```

```

2024-05-01T15:37:03.271+05:30 INFO 46871 --- [main] c.c.l.SpringbootApplication
2024-05-01T15:37:03.272+05:30 INFO 46871 --- [main] c.c.l.SpringbootApplication
2024-05-01T15:37:03.710+05:30 INFO 46871 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer
2024-05-01T15:37:03.714+05:30 INFO 46871 --- [main] o.apache.catalina.core.StandardService
2024-05-01T15:37:03.714+05:30 INFO 46871 --- [main] o.apache.catalina.core.StandardEngine
2024-05-01T15:37:03.739+05:30 INFO 46871 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/]
2024-05-01T15:37:03.740+05:30 INFO 46871 --- [main] w.s.c.ServletWebServerApplicationContext
User initialization
User object hashCode: 254812619
2024-05-01T15:37:03.926+05:30 INFO 46871 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer
2024-05-01T15:37:03.926+05:30 INFO 46871 --- [main] c.c.l.SpringbootApplication
2024-05-01T15:37:03.710+05:30 INFO 46871 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer
2024-05-01T15:37:03.714+05:30 INFO 46871 --- [main] o.apache.catalina.core.StandardService
2024-05-01T15:37:03.714+05:30 INFO 46871 --- [main] o.apache.catalina.core.StandardEngine
2024-05-01T15:37:03.739+05:30 INFO 46871 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/]
2024-05-01T15:37:03.740+05:30 INFO 46871 --- [main] w.s.c.ServletWebServerApplicationContext
User initialization
User object hashCode: 254812619
2024-05-01T15:37:03.926+05:30 INFO 46871 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer
2024-05-01T15:37:03.926+05:30 INFO 46871 --- [main] c.c.l.SpringbootApplication
2024-05-01T15:37:03.710+05:30 INFO 46871 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/]
2024-05-01T15:37:03.714+05:30 INFO 46871 --- [main] o.s.web.servlet.DispatcherServlet
2024-05-01T15:37:03.714+05:30 INFO 46871 --- [main] o.s.web.servlet.DispatcherServlet
TestController1 instance initialization
TestController1 object hashCode: 1478378887 User object hashCode: 254812619
fetchUser api invoked
1. ① localhost:8090/api/fetchUser
```

```

2024-05-01T15:37:03.271+05:30 INFO 46871 --- [main] c.c.l.SpringbootApplication
2024-05-01T15:37:03.272+05:30 INFO 46871 --- [main] c.c.l.SpringbootApplication
2024-05-01T15:37:03.710+05:30 INFO 46871 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer
2024-05-01T15:37:03.714+05:30 INFO 46871 --- [main] o.apache.catalina.core.StandardService
2024-05-01T15:37:03.714+05:30 INFO 46871 --- [main] o.apache.catalina.core.StandardEngine
2024-05-01T15:37:03.739+05:30 INFO 46871 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/]
2024-05-01T15:37:03.740+05:30 INFO 46871 --- [main] w.s.c.ServletWebServerApplicationContext
User initialization
User object hashCode: 254812619
2024-05-01T15:37:03.926+05:30 INFO 46871 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer
2024-05-01T15:37:03.926+05:30 INFO 46871 --- [main] c.c.l.SpringbootApplication
2024-05-01T15:37:03.710+05:30 INFO 46871 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/]
2024-05-01T15:37:03.714+05:30 INFO 46871 --- [main] o.s.web.servlet.DispatcherServlet
2024-05-01T15:37:03.714+05:30 INFO 46871 --- [main] o.s.web.servlet.DispatcherServlet
TestController1 instance initialization
TestController1 object hashCode: 1478378887 User object hashCode: 254812619
fetchUser api invoked
2. ① localhost:8090/api/fetchUser
```

```

3. ① localhost:8090/api/logout
2024-05-01T17:44:42.588+05:30 INFO 52913 --- [main] w.s.c.ServletWebServerApplicationContext
User initialization
User object hashCode: 254812619
2024-05-01T17:44:42.588+05:30 INFO 52913 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer
2024-05-01T17:44:42.590+05:30 INFO 52913 --- [main] c.c.l.SpringbootApplication
2024-05-01T17:44:42.590+05:30 INFO 52913 --- [nio-8890-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]
2024-05-01T17:44:42.590+05:30 INFO 52913 --- [nio-8890-exec-1] o.s.web.servlet.DispatcherServlet
2024-05-01T17:44:42.590+05:30 INFO 52913 --- [nio-8890-exec-1] o.s.web.servlet.DispatcherServlet
TestController1 instance initialization
TestController1 object hashCode: 460533944 User object hashCode: 254812619
fetchUser api invoked
fetchUser api invoked
end the session
4. ① localhost:8090/api/fetchUser
2024-05-01T17:44:42.588+05:30 INFO 52913 --- [main] w.s.c.ServletWebServerApplicationContext
User initialization
User object hashCode: 254812619
2024-05-01T17:44:42.588+05:30 INFO 52913 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer
2024-05-01T17:44:42.588+05:30 INFO 52913 --- [main] c.c.l.SpringbootApplication
2024-05-01T17:44:42.588+05:30 INFO 52913 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/]
2024-05-01T17:44:42.588+05:30 INFO 52913 --- [nio-8890-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]
2024-05-01T17:44:42.588+05:30 INFO 52913 --- [nio-8890-exec-1] o.s.web.servlet.DispatcherServlet
2024-05-01T17:44:42.588+05:30 INFO 52913 --- [nio-8890-exec-1] o.s.web.servlet.DispatcherServlet
TestController1 instance initialization
TestController1 object hashCode: 460533944 User object hashCode: 254812619
fetchUser api invoked
fetchUser api invoked
TestController1 instance initialization
TestController1 object hashCode: 754846954 User object hashCode: 254812619
fetchUser api invoked

```

## Sprigboot: Dynamic Bean Initialization

**Unsatisfied Dependency problem**

The screenshot shows a Java application interface with several code snippets and error messages.

**User Controller Code:**

```
@RestController
@RequestMapping(value = "/api")
public class User {
    @Autowired
    Order order;

    @PostMapping("/createOrder")
    public ResponseEntity<String> createOrder() {
        order.createOrder();
        return ResponseEntity.ok("body");
    }
}
```

**Order Interface:**

```
public interface Order {
    public void createOrder();
}
```

**OnlineOrder Implementation:**

```
@Component
public class OnlineOrder implements Order {
    public OnlineOrder(){
        System.out.println("Online Order Initialized");
    }

    public void createOrder(){
        System.out.println("created Online Order");
    }
}
```

**OfflineOrder Implementation:**

```
@Component
public class OfflineOrder implements Order {
    public OfflineOrder(){
        System.out.println("Offline Order initialized");
    }

    public void createOrder(){
        System.out.println("created Offline Order");
    }
}
```

**Error Message:**

```
*****
APPLICATION FAILED TO START
*****
UnsatisfiedDependencyException: Error creating bean with name 'User'
```

**@Qualifier**

```

@RestController
@RequestMapping(value = "/api")
public class User {

    @Qualifier("onlineOrderObject")
    @Autowired
    Order order;

    @PostMapping("createOrder")
    public ResponseEntity<String> createOrder() {
        order.createOrder();
        return ResponseEntity.ok(body: "");
    }
}

public interface Order {
    public void createOrder();
}

@Qualifier("onlineOrderObject")
@Component
public class OnlineOrder implements Order{
    public OnlineOrder(){
        System.out.println("Online Order Initialized");
    }

    public void createOrder(){
        System.out.println("created Online Order");
    }
}

@Qualifier("offlineOrderObject")
@Component
public class OfflineOrder implements Order{
    public OfflineOrder(){
        System.out.println("Offline Order initialized");
    }

    public void createOrder(){
        System.out.println("created Offline Order");
    }
}

```

2024-05-07T16:41:14.520+05:30 INFO 98826 --- [main] c.c.l.SpringbootApplication
2024-05-07T16:41:14.521+05:30 INFO 98826 --- [main] c.c.l.SpringbootApplication
2024-05-07T16:41:14.959+05:30 INFO 98826 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer
2024-05-07T16:41:14.959+05:30 INFO 98826 --- [main] o.apache.catalina.core.StandardService
2024-05-07T16:41:14.959+05:30 INFO 98826 --- [main] o.apache.catalina.core.StandardEngine
2024-05-07T16:41:14.985+05:30 INFO 98826 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/]
2024-05-07T16:41:14.985+05:30 INFO 98826 --- [main] w.s.c.ServletWebServerApplicationContext
Offline Order initialized
Online Order Initialized
2024-05-07T16:41:15.149+05:30 INFO 98826 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer
2024-05-07T16:41:15.154+05:30 INFO 98826 --- [main] c.c.l.SpringbootApplication

POST http://localhost:8080/api/createOrder

```

2024-05-07T16:41:14.959+05:30 INFO 98826 --- [main] o.apache.catalina.core.StandardEngine
2024-05-07T16:41:14.985+05:30 INFO 98826 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/]
2024-05-07T16:41:14.985+05:30 INFO 98826 --- [main] w.s.c.ServletWebServerApplicationContext
Offline Order initialized
Online Order Initialized
2024-05-07T16:41:15.149+05:30 INFO 98826 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer
2024-05-07T16:41:15.154+05:30 INFO 98826 --- [main] c.c.l.SpringbootApplication
2024-05-07T16:41:15.154+05:30 INFO 98826 --- [nio-8080-exec-2] o.a.c.c.C.[Tomcat].[localhost].[/]
2024-05-07T16:41:15.154+05:30 INFO 98826 --- [nio-8080-exec-2] o.s.web.DispatcherServlet
2024-05-07T16:41:15.154+05:30 INFO 98826 --- [nio-8080-exec-2] o.s.web.DispatcherServlet
created Online Order

```

**1st Solution:**

```

@RestController
@RequestMapping(value = "/api")
public class User {

    @Qualifier("onlineOrderObject")
    @Autowired
    Order onlineOrder;

    @Qualifier("offlineOrderObject")
    @Autowired
    Order offlineOrder;

    @PostMapping("createOrder")
    public ResponseEntity<String> createOrder(@Qualifier("onlineOrder") Order onlineOrder) {
        if (onlineOrder != null) {
            onlineOrder.createOrder();
        } else {
            offlineOrder.createOrder();
        }
        return ResponseEntity.ok(body: "");
    }
}

public interface Order {
    public void createOrder();
}

@Qualifier("onlineOrderObject")
@Component
public class OnlineOrder implements Order{
    public OnlineOrder(){
        System.out.println("Online Order Initialized");
    }

    public void createOrder(){
        System.out.println("created Online Order");
    }
}

@Qualifier("offlineOrderObject")
@Component
public class OfflineOrder implements Order{
    public OfflineOrder(){
        System.out.println("Offline Order initialized");
    }

    public void createOrder(){
        System.out.println("created Offline Order");
    }
}

```

## 2nd Solution:

```
@RestController
@RequestMapping(value = "/api")
public class User {

    @Autowired
    Order order;

    @PostMapping("/createOrder")
    public ResponseEntity<String> createOrder() {
        order.createOrder();
        return ResponseEntity.ok("body: ");
    }
}

@Configuration
public class AppConfig {

    @Bean
    public Order createOrder(@Value("${isOnlineOrder}") boolean isOnlineOrder) {
        if(isOnlineOrder) {
            return new OnlineOrder();
        } else {
            return new OfflineOrder();
        }
    }
}
```

```
public interface Order {
    public void createOrder();
}

public class OnlineOrder implements Order{
    public OnlineOrder(){
        System.out.println("Online Order Initialized");
    }

    public void createOrder(){
        System.out.println("created Online Order");
    }
}

public class OfflineOrder implements Order{
    public OfflineOrder(){
        System.out.println("Offline Order initialized");
    }

    public void createOrder(){
        System.out.println("created Offline Order");
    }
}
```

```
application.properties
isOnlineOrder=false
```

```
2024-05-08T19:30:38.605+05:30 INFO 45167 --- [           main] o.apache.catalina.core.StandardService   : 
2024-05-08T19:30:38.606+05:30 INFO 45167 --- [           main] o.apache.catalina.core.StandardEngine    : 
2024-05-08T19:30:38.631+05:30 INFO 45167 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[]         : 
2024-05-08T19:30:38.632+05:30 INFO 45167 --- [           main] w.s.c.ServletWebServerApplicationContext  : 
Offline Order initialized
2024-05-08T19:30:38.793+05:30 INFO 45167 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : 
2024-05-08T19:30:38.798+05:30 INFO 45167 --- [           main] c.c.l.SpringbootApplication               : 
```

```
POST      http://localhost:8090/api/createOrder
```

```
2024-05-08T19:30:38.606+05:30 INFO 45167 --- [           main] o.apache.catalina.core.StandardEngine   : 
2024-05-08T19:30:38.631+05:30 INFO 45167 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[]       : 
2024-05-08T19:30:38.632+05:30 INFO 45167 --- [           main] w.s.c.ServletWebServerApplicationContext : 
Offline Order initialized
2024-05-08T19:30:38.793+05:30 INFO 45167 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : 
2024-05-08T19:30:38.798+05:30 INFO 45167 --- [           main] c.c.l.SpringbootApplication             : 
2024-05-08T19:32:33.610+05:30 INFO 45167 --- [nio-8090-exec-2] o.a.c.c.C.[Tomcat].[localhost].[]       : 
2024-05-08T19:32:33.610+05:30 INFO 45167 --- [nio-8090-exec-2] o.s.web.servlet.DispatcherServlet      : 
2024-05-08T19:32:33.611+05:30 INFO 45167 --- [nio-8090-exec-2] o.s.web.servlet.DispatcherServlet      : 
created Offline Order
```

### **@Value :**

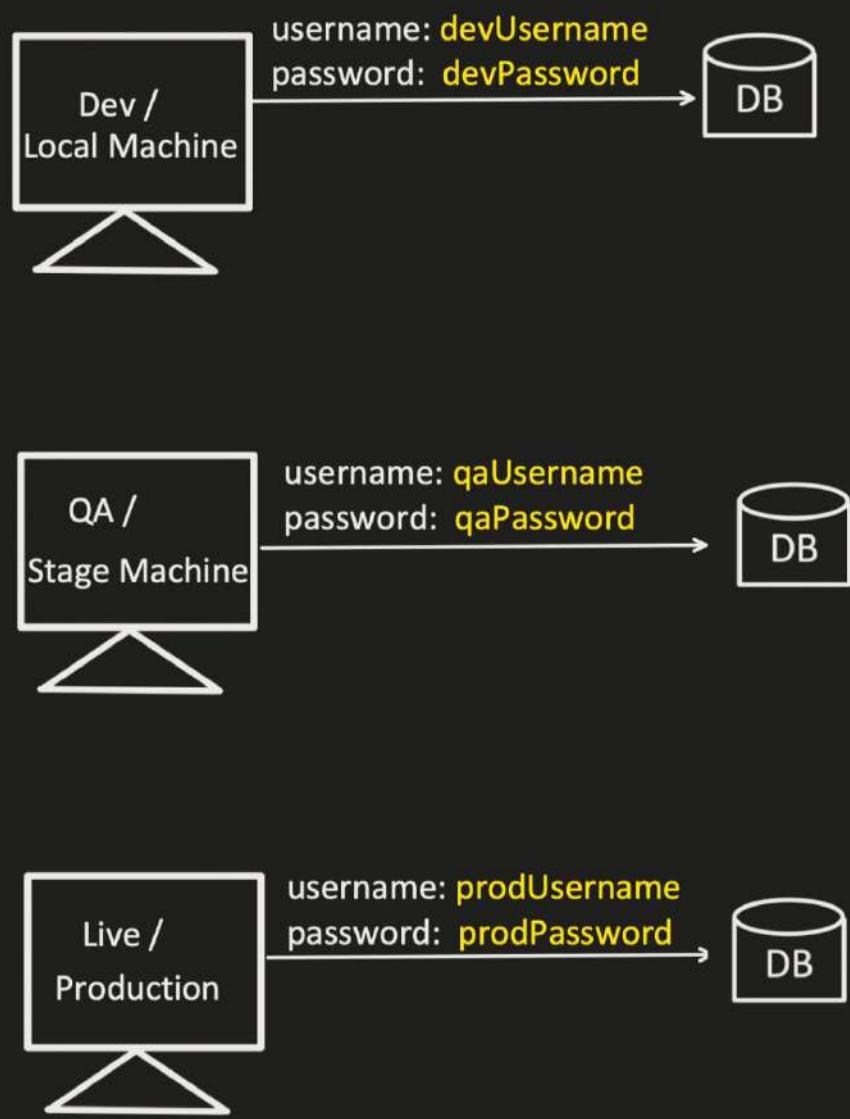
it is used to inject values from various sources like property file, environment variables or inline literals.

#### *Inline Literal Example*

```
@Bean
public Order createOrderBean(@Value("false") boolean isOnlineOrder){
    if(isOnlineOrder) {
        return new OnlineOrder();
    } else {
        return new OfflineOrder();
    }
}
```

## Spring boot: @Profile

Let's deep dive into **@Profile** annotation to understand it better



This "username", "password" is just one example.

There are so many other configuration, which are different for different environments, like:

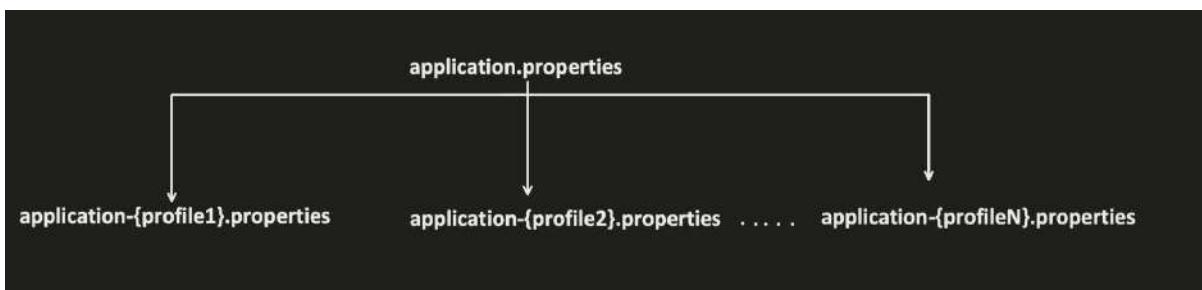
- URL and Port number
- Connection timeout values
- Request timeout values
- Throttle values
- Retry values etc.

### How to do it?

We put the configurations in "application.properties" file.

But how to handle, different environment configurations?

That's where Profiling comes into the picture



**application.properties**

```
username=defaultUsername
password=defaultPassword
```

**application-dev.properties**

```
username=devUsername
password=devPassword
```

**application-qa.properties**

```
username=qaUsername
password=qaPassword
```

**application-prod.properties**

```
username=prodUsername
password=prodPassword
```

```

2024-06-01T22:55:24.548+05:30 INFO 48119 --- [main] c.c.l.SpringbootApplication : The following 1 profile is active: "qa"
2024-06-01T22:55:24.549+05:30 INFO 48119 --- [main] c.c.l.SpringbootApplication : Tomcat initialized with port: 8080 (http)
2024-06-01T22:55:24.982+05:30 INFO 48119 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Starting service [Tomcat]
2024-06-01T22:55:24.990+05:30 INFO 48119 --- [main] o.apache.catalina.core.StandardService : Starting Servlet Engine: [Apache Tomcat/10.1.19]
2024-06-01T22:55:24.999+05:30 INFO 48119 --- [main] o.apache.catalina.core.StandardEngine : [Tomcat].([localhost]).[/]
2024-06-01T22:55:25.016+05:30 INFO 48119 --- [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 625 ms
2024-06-01T22:55:25.016+05:30 INFO 48119 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port: 8080 (http) with context path ''
2024-06-01T22:55:25.179+05:30 INFO 48119 --- [main] c.c.l.SpringbootApplication : Started SpringBootApplication in 0.76 seconds (process running for 0.995)

```

And during application startup, we can tell spring boot to pick specific "application properties" file, Using "spring.profiles.active" configuration.

**application.properties**

```
username=defaultUsername
password=defaultPassword
spring.profiles.active=qa
```

**application-dev.properties**

```
username=devUsername
password=devPassword
```

**application-qa.properties**

```
username=qaUsername
password=qaPassword
```

**application-prod.properties**

```
username=prodUsername
password=prodPassword
```

```

2024-06-02T08:04:23.509+05:30 INFO 52375 --- [main] c.c.l.SpringbootApplication : The following 1 profile is active: "qa"
2024-06-02T08:04:23.509+05:30 INFO 52375 --- [main] c.c.l.SpringbootApplication : Tomcat initialized with port: 8080 (http)
2024-06-02T08:04:24.409+05:30 INFO 52375 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Starting service [Tomcat]
2024-06-02T08:04:24.409+05:30 INFO 52375 --- [main] o.apache.catalina.core.StandardService : Starting Servlet Engine: [Apache Tomcat/10.1.19]
2024-06-02T08:04:24.419+05:30 INFO 52375 --- [main] o.apache.catalina.core.StandardEngine : [Tomcat].([localhost]).[/]
2024-06-02T08:04:24.419+05:30 INFO 52375 --- [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 625 ms
2024-06-02T08:04:24.419+05:30 INFO 52375 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port: 8080 (http) with context path ''
2024-06-02T08:04:24.419+05:30 INFO 52375 --- [main] c.c.l.SpringbootApplication : Started SpringBootApplication in 0.76 seconds (process running for 0.995)

```

We can pass the value to this configuration "spring.profiles.active" during application startup itself.

```
mvn spring-boot:run -Dspring-boot.run.profiles=prod
```

or

Add this in Pom.xml

```
<profiles>
    <profile>
        <id>local</id>
        <properties>
            <spring-boot.run.profiles>dev</spring-boot.run.profiles>
        </properties>
    </profile>
    <profile>
        <id>production</id>
        <properties>
            <spring-boot.run.profiles>prod</spring-boot.run.profiles>
        </properties>
    </profile>
    <profile>
        <id>stage</id>
        <properties>
            <spring-boot.run.profiles>qa</spring-boot.run.profiles>
        </properties>
    </profile>
</profiles>
```

```
2024-06-02T08:21:57.336+05:30 INFO 53896 --- [main] c.c.l.SpringbootApplication : The following 1 profile is active: "prod"
2024-06-02T08:21:57.647+05:30 INFO 53896 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port 8080 (http)
2024-06-02T08:21:57.652+05:30 INFO 53896 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2024-06-02T08:21:57.652+05:30 INFO 53896 --- [main] o.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/10.1.19]
2024-06-02T08:21:57.673+05:30 INFO 53896 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2024-06-02T08:21:57.673+05:30 INFO 53896 --- [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 318 ms
username:prodUsername password: prodPassword
2024-06-02T08:21:57.804+05:30 INFO 53896 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8080 (http) with context path ''

```

```
mvn spring-boot:run -Pproduction
```

So, now we know, what's Profiling is. Lets see what is @Profile annotation.

Using @Profile annotation, we can tell spring boot, to create bean only when particular profile is set.

```
@Component
@Profile("prod")
public class MySQLConnection {

    @Value("${username}")
    String username;

    @Value("${password}")
    String password;

    @PostConstruct
    public void init() {
        System.out.println("MySQL username: " + username + " password: " + password);
    }
}
```

```
@Component
@Profile("dev")
public class MySQLConnection {

    @Value("${username}")
    String username;

    @Value("${password}")
    String password;

    @PostConstruct
    public void init() {
        System.out.println("MySQL username: " + username + " password: " + password);
    }
}
```

application.properties

```
username=defaultUsername
password=defaultPassword
spring.profiles.active=qa
```

application-dev.properties      application-qa.properties      application-prod.properties

```
username=devUsername
password=devPassword
```

```
username=qaUsername
password=qaPassword
```

```
username=prodUsername
password=prodPassword
```

```

mvn spring-boot:run

2024-06-02T00:38:46.935+05:30 INFO 55048 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port 8080 (http)
2024-06-02T00:38:46.940+05:30 INFO 55048 --- [           main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2024-06-02T00:38:46.941+05:30 INFO 55048 --- [           main] o.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/10.1.19]
2024-06-02T00:38:46.964+05:30 INFO 55048 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2024-06-02T00:38:46.966+05:30 INFO 55048 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 410 ms
2024-06-02T00:38:47.109+05:30 INFO 55048 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8080 (http) with context path ''


mvn spring-boot:run -Dspring-boot.run.profiles=prod

2024-06-02T00:42:39.245+05:30 INFO 55216 --- [           main] o.c.l.SpringBootApplication : The following 1 profile is active: "prod"
2024-06-02T00:42:39.563+05:30 INFO 55216 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port 8080 (http)
2024-06-02T00:42:39.568+05:30 INFO 55216 --- [           main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2024-06-02T00:42:39.568+05:30 INFO 55216 --- [           main] o.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/10.1.19]
2024-06-02T00:42:39.591+05:30 INFO 55216 --- [           main] o.a.c.c.C.[Tomcat].[localhost]. [/] : Initializing Spring embedded WebApplicationContext
2024-06-02T00:42:39.591+05:30 INFO 55216 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 327 ms
MySQL username: produsername password: prodpassword
2024-06-02T00:42:39.731+05:30 INFO 55216 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8080 (http) with context path ''
```

We can also set multiple profiles at a time.

```

@Component
@Profile("prod")
public class MySQLConnection {

    @Value("${username}")
    String username;

    @Value("${password}")
    String password;

    @PostConstruct
    public void init() {
        System.out.println("MySQL username: " + username + " password: " + password);
    }
}

```

**application.properties**

```

username=defaultUsername
password=defaultPassword
spring.profiles.active=prod,qa

```

application-dev.properties	application-qa.properties	application-prod.properties
username=devUsername password=devPassword	username=qaUsername password=qaPassword	username=prodUsername password=prodPassword

```

2024-06-02T00:45:26.744+05:30 INFO 56036 --- [           main] o.c.l.SpringBootApplication : The following 2 profiles are active: "prod", "qa"
2024-06-02T00:45:27.514+05:30 INFO 56036 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port 8080 (http)
2024-06-02T00:45:27.534+05:30 INFO 56036 --- [           main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2024-06-02T00:45:27.535+05:30 INFO 56036 --- [           main] o.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/10.1.19]
2024-06-02T00:45:27.539+05:30 INFO 56036 --- [           main] o.a.c.c.C.[Tomcat].[localhost]. [/] : Initializing Spring embedded WebApplicationContext
2024-06-02T00:45:27.539+05:30 INFO 56036 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 453 ms
MySQL username: qausername password: qaPassword
2024-06-02T00:45:27.539+05:30 INFO 56036 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8080 (http) with context path ''
```

Now lets come back to the previous question, which I asked:

You have 2 Application and 1 common code base, how you will make sure that, BEAN is only created for 1 Application, not for other?

### Common Codebase

```
@Component  
@Profile("app1")  
public class NoSQLConnection {  
  
    @Value("${username}")  
    String username;  
  
    @Value("${password}")  
    String password;  
  
    @PostConstruct  
    public void init(){  
        System.out.println("NoSQL username: " + username + " password: " + password);  
    }  
}
```

Application1  
(application.properties)

```
spring.profiles.active=app1
```

Application2  
(application.properties)

```
spring.profiles.active=app2
```

## Spring boot: @ConditionalOnProperty

`@ConditionalOnProperty :`

Bean is created Conditionally (mean Bean can be created or Not).

We have already know the behavior of below program:

```
@Component
public class DBConnection {

    @Autowired
    MySQLConnection mySQLConnection;

    @Autowired
    NoSQLConnection noSQLConnection;

    @PostConstruct
    public void init(){
        System.out.println("DB Connection Bean Created with dependencies below:");
        System.out.println("is MySQLConnection object Null: " + Objects.isNull(mySQLConnection));
        System.out.println("is NoSQLConnection object Null: " + Objects.isNull(noSQLConnection));
    }
}

@Component
public class NoSQLConnection {

    NoSQLConnection() {
        System.out.println("Initialization of NoSQLConnection Bean");
    }
}

@Component
public class MySQLConnection {

    MySQLConnection(){
        System.out.println("Initialization of MySQLConnection Bean");
    }
}
```

2024-05-25T10:55:57.207+05:30 INFO 6700 --- [main] w.s.c.ServletWebServerApplicationContext initialization of MySQLConnection Bean  
initialization of NoSQLConnection Bean  
DB Connection Bean Created with dependencies below:  
is MySQLConnection object Null: false  
is NoSQLConnection object Null: false  
2024-05-25T10:55:57.381+05:30 INFO 6700 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer  
2024-05-25T10:55:57.387+05:30 INFO 6700 --- [main] c.c.l.SpringbootApplication

But how you will handle below Use Cases?

Use case 1:

We want to create only 1 Bean, either *MySQLConnection* or *NoSQLConnection*.

Use case 2:

We have 2 components, sharing same codebase, But 1 component need *MySQLConnection* and other needs *NoSQLConnection*

Solution is @ConditionalOnProperty Annotation

```

@Component
public class DBConnection {
    @Autowired(required = false)
    MySQLConnection mySQLConnection;

    @Autowired(required = false)
    NoSQLConnection noSQLConnection;

    @PostConstruct
    public void init() {
        System.out.println("DB Connection Bean Created with dependencies below:");
        System.out.println("is MySQLConnection object Null: " + Objects.isNull(mySQLConnection));
        System.out.println("is NoSQLConnection object Null: " + Objects.isNull(noSQLConnection));
    }
}

@Component
@ConditionalOnProperty(prefix = "sqlconnection", value = "enabled", havingValue = "true", matchIfMissing = false)
public class MySQLConnection {
    MySQLConnection() {
        System.out.println("Initialization of MySQLConnection Bean");
    }
}

@Component
@ConditionalOnProperty(prefix = "nosqlconnection", value = "enabled", havingValue = "true", matchIfMissing = false)
public class NoSQLConnection {
    NoSQLConnection() {
        System.out.println("Initialization of NoSQLConnection Bean");
    }
}

```

Application.properties

```

sqlconnection.enabled=true

```

```

2024-05-25T11:32:03.950+05:30 INFO 8352 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/]
2024-05-25T11:32:03.951+05:30 INFO 8352 --- [main] w.s.c.ServletWebServerApplicationContext
initialization of MySQLConnection Bean
DB Connection Bean Created with dependencies below:
is MySQLConnection object Null: false
is NoSQLConnection object Null: true
2024-05-25T11:32:04.102+05:30 INFO 8352 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer
2024-05-25T11:32:04.106+05:30 INFO 8352 --- [main] c.c.l.SpringbootApplication

```

### Advantages:

1. Toggling of Feature
2. Avoid cluttering Application context with un-necessary beans.
3. Save Memory
4. Reduce application Startup time

### Disadvantages:

1. Misconfiguration can happen.
2. Code Complexity when over used.
3. Multiple bean creation with same Configuration, brings confusion.
4. Complexity in managing.

## Spring boot : AOP

### AOP (Aspect Oriented Programming)

- In simple term, It helps to Intercept the method invocation. And we can perform some task before and after the method.
- AOP allow us to focus on business logic by handling boilerplate and repetitive code like logging, transaction management etc.
- So, Aspect is a module which handle this repetitive or boilerplate code.
- Helps in achieving reusability, maintainability of the code.

Used during:

- Logging
- Transaction Management
- Security etc..

### Dependency you need to add in pom.xml:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```

```

@RestController
@RequestMapping(value = "/api/")
public class Employee {

    @GetMapping (path = "/fetchEmployee")
    public String fetchEmployee(){
        return "item fetched";
    }
}

```

```

@Component
@Aspect
public class LoggingAspect {

    @Before("execution(public String com.conceptandcoding.learningspringboot.Employee.fetchEmployee())")
    public void beforeMethod(){
        System.out.println("inside beforeMethod Aspect");
    }
}

```

```

2024-06-16T23:03:38.117+05:30 INFO 18766 --- [main] org.hibernate.Version : HH0080412: Hibernate ORM core version 6.4.4.Final
2024-06-16T23:03:38.157+05:30 INFO 18766 --- [main] o.h.c.internal.RegionFactoryInitiator : HH0080026: Second-level cache disabled
2024-06-16T23:03:38.237+05:30 INFO 18766 --- [main] o.s.e.j.p.SpringPersistenceUnitInfo : No LoadTimeWeaver setup: ignoring JPA class transformer
2024-06-16T23:03:38.359+05:30 INFO 18766 --- [main] o.h.e.t.j.p.l.JtaPlatformInitiator : HH0080409: No JTA platforms available (set 'hibernate.transaction.jta.platforms')
2024-06-16T23:03:38.400+05:30 INFO 18766 --- [main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
2024-06-16T23:03:38.440+05:30 WARN 18766 --- [main] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database queries may
2024-06-16T23:03:38.632+05:30 INFO 18766 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8080 (http) with context path ''
2024-06-16T23:03:38.637+05:30 INFO 18766 --- [main] c.c.l.SpringbootApplication : Started SpringbootApplication in 1.517 seconds (process running for 1.727)

```

```

<--> C localhost:8080/api/fetchEmployee
item fetched

```

```

2024-06-16T23:03:38.237+05:30 INFO 18766 --- [main] o.s.e.j.p.SpringPersistenceUnitInfo : No LoadTimeWeaver setup: ignoring JPA class transformer
2024-06-16T23:03:38.398+05:30 INFO 18766 --- [main] o.h.e.t.j.p.l.JtaPlatformInitiator : HH0080409: No JTA platform available (set 'hibernate.transaction.jta.platforms')
2024-06-16T23:03:38.400+05:30 INFO 18766 --- [main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
2024-06-16T23:03:38.532+05:30 WARN 18766 --- [main] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database queries may
2024-06-16T23:03:38.537+05:30 INFO 18766 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8080 (http) with context path ''
2024-06-16T23:03:38.497+05:30 INFO 18766 --- [nio-8088-exec-1] o.a.c.c.C.[localhost].[/] : Initializing Spring DispatcherServlet 'dispatcherServlet'
2024-06-16T23:03:38.498+05:30 INFO 18766 --- [nio-8088-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
2024-06-16T23:03:38.499+05:30 INFO 18766 --- [nio-8088-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 1 ms
Inside beforeMethod Aspect

```

**Some important AOP concepts :**

```

@Component
@Aspect
public class LoggingAspect {

    @Before("execution(public String com.conceptandcoding.learningspringboot.Employee.fetchEmployee())")
    public void beforeMethod(){
        System.out.println("inside beforeMethod Aspect");
    }
}

```

Access-modifiers: optional and can be omitted

Return type: optional but can not be omitted

This is a pointcut

This @Before and method together is called Advice

**Pointcut:**  
Its an Expression, which tells where an ADVICE should be applied.

**Types of Pointcut:**

1. Execution: matches a particular method in a particular class.

```
@Before("execution(public String com.conceptandcoding.learningspringboot.Employee.fetchEmployee())")
```

→ (\*) wildcard : matches any single item

Matches any return type

```
@Before("execution(* com.conceptandcoding.learningspringboot.Employee.fetchEmployee())")
public void beforeMethod(){
    System.out.println("inside beforeMethod Aspect");
}
```

Matches any method with single parameter String

```
@Before("execution(* com.conceptandcoding.learningspringboot.Employee.*(String))")
public void beforeMethod(){
    System.out.println("inside beforeMethod Aspect");
}
```

Matches fetchEmployee method that take any single parameter

```
@Before("execution(String com.conceptandcoding.learningspringboot.Employee.fetchEmployee(..))")
public void beforeMethod(){
    System.out.println("inside beforeMethod Aspect");
}
```

→ (..) wildcard : matches 0 or More item

Matches fetchEmployee method that take any 0 or More parameters

```
@Before("execution(String com.conceptandcoding.learningspringboot.Employee.fetchEmployee(..))")
public void beforeMethod(){
    System.out.println("inside beforeMethod Aspect");
}
```

Matches fetchEmployee method in 'com.conceptandcoding' package and subpackage classes

```
@Before("execution(String com.conceptandcoding..fetchEmployee(..))")
public void beforeMethod(){
    System.out.println("inside beforeMethod Aspect");
}
```

Matches any method in 'com.conceptandcoding' package and subpackage classes

```
@Before("execution(String com.conceptandcoding..*(..))")
public void beforeMethod(){
    System.out.println("inside beforeMethod Aspect");
}
```

2. Within: matches all method within any class or package.

- This pointcut will run for each method in the class Employee  
`@Before("within(com.conceptandcoding.learningspringboot.Employee)")`
- This pointcut will run for each method in this package and subpackage  
`@Before("within(com.conceptandcoding.learningspringboot..*)")`

3. @within: matches any method in a class which has this annotation.

```
@RestController
@RequestMapping(value = "/api/")
public class Employee {

    @Autowired
    EmployeeUtil employeeUtil;

    @GetMapping (path = "/fetchEmployee")
    public String fetchEmployee(){
        employeeUtil.employeeHelperMethod();
        return "item fetched";
    }
}
```

```
@Aspect
@Component
public class LoggingAspect {

    @Before("@within(org.springframework.stereotype.Service)")
    public void beforeMethod() {
        System.out.println("inside beforeMethod aspect");
    }

    @Service
    public class EmployeeUtil {

        public void employeeHelperMethod() {
            System.out.println("employee helper method called");
        }
    }
}
```

```
2024-06-22T11:54:02,459+05:30 INFO 66918 --- [           main] c.c.l.SpringbootApplication          : Started SpringbootApplication in 1.646s
2024-06-22T11:54:04,070+05:30 INFO 66918 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[]   : Initializing Spring DispatcherServlet
2024-06-22T11:54:04,070+05:30 INFO 66918 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet      : Initializing Servlet 'dispatcherServlet'
2024-06-22T11:54:04,070+05:30 INFO 66918 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet      : Completed initialization in 0 ms
inside beforeMethod aspect
employee helper method called
```

**4. @annotation: matches any method that is annotated with given annotation**

```
@RestController
@RequestMapping(value = "/api/")
public class Employee {

    @GetMapping (path = "/fetchEmployee")
    public String fetchEmployee(){
        return "item fetched";
    }
}
```

```
@Component
@Aspect
public class LoggingAspect {

    @Before("@annotation(org.springframework.web.bind.annotation.GetMapping)")
    public void beforeMethod(){
        System.out.println("inside beforeMethod Aspect");
    }
}
```

**5. Args: matches any method with particular arguments (or parameters)**

```
@Before("args(String,int)")
```

```
@RestController
@RequestMapping(value = "/api/")
public class Employee {

    @Autowired
    EmployeeUtil employeeUtil;

    @GetMapping (path = "/fetchEmployee")
    public String fetchEmployee(){
        employeeUtil.employeeHelperMethod( str: "xyz", val: 123);
        return "item fetched";
    }
}
```

```
@Aspect
@Component
public class LoggingAspect {

    @Before("args(String, int)")
    public void beforeMethod() {
        System.out.println("inside beforeMethod aspect");
    }
}

@Service
public class EmployeeUtil {

    public void employeeHelperMethod(String str, int val) {
        System.out.println("employee helper method called");
    }
}
```

```
2024-06-22T11:13:27.258+05:30 INFO 44894 --- [main] o.hibernate.jpa.internal.util.LogHelper : HHH000204: Processing PersistenceUnitInfo
2024-06-22T11:13:27.290+05:30 INFO 44894 --- [main] org.hibernate.Version : HHH000412: Hibernate ORM core
2024-06-22T11:13:27.309+05:30 INFO 44894 --- [main] o.h.c.internal.RegionFactoryInitiator : HHH000806: Second-Level cache
2024-06-22T11:13:27.414+05:30 INFO 44894 --- [main] o.s.o.j.p.SpringPersistenceUnitInfo : No LoadTimeWeaver setup; ignore
2024-06-22T11:13:27.503+05:30 INFO 44894 --- [main] o.h.e.t.p.i.JtaPlatformInitiator : HHH000809: No JTA platform available
2024-06-22T11:13:27.565+05:30 INFO 44894 --- [main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'spring'
2024-06-22T11:13:27.651+05:30 WARN 44894 --- [main] jpaBaseConfiguration$JpaVendorAdapter : spring.jpa.open-in-view is enabled by default. Add 'spring.open-in-view' to the configuration to disable.
2024-06-22T11:13:27.804+05:30 INFO 44894 --- [main] o.n.o.v.rebindable.tenant.TenantRebindServer : Tenant started on port 8080
2024-06-22T11:13:27.874+05:30 INFO 44894 --- [main] c.c.l.SpringbootApplication : Started SpringbootApplication
2024-06-22T11:13:30.484+05:30 INFO 44894 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcher'
2024-06-22T11:13:30.489+05:30 INFO 44894 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 1
inside beforeMethod aspect
employee helper method called
```

If instead of primitive type, we need object, then we can give like this

```
@Before("args(com.conceptandcoding.learningspringboot.Employee)")
```

**6. @args: matches any method with particular parameters and that parameter class is annotated with particular annotation.**

```
@Before("@args(org.springframework.stereotype.Service)")
```

```
@Service
public class EmployeeUtil {

    public void employeeHelperMethod(EmployeeDAO employeeDAO) {
        System.out.println("employee helper method called");
    }
}
```

```
@Aspect
@Component
public class LoggingAspect {

    @Before("@args(org.springframework.stereotype.Service)")
    public void beforeMethod() {
        System.out.println("inside beforeMethod aspect");
    }
}

@Service
public class EmployeeDAO {
```

7. target: matches any method on a particular instance of a class.

```
@RestController  
@RequestMapping(value = "/api/")  
public class Employee {  
  
    @Autowired  
    EmployeeUtil employeeUtil;  
  
    @GetMapping (path = "/fetchEmployee")  
    public String fetchEmployee(){  
        employeeUtil.employeeHelperMethod();  
        return "item fetched";  
    }  
}
```

```
@Aspect  
@Component  
public class LoggingAspect {  
  
    @Before("target(com.conceptandcoding.learningspringboot.EmployeeUtil)")  
    public void beforeMethod() {  
        System.out.println("inside beforeMethod aspect");  
    }  
}
```

```
@Component  
public class EmployeeUtil {  
  
    public void employeeHelperMethod() {  
        System.out.println("employee helper method called");  
    }  
}
```

Interface

```

    @Before("target(com.conceptandcoding.learningspringboot.IEmployee)")

    @RestController
    @RequestMapping(value = "/api/")
    public class EmployeeController {

        @Autowired
        @Qualifier("tempEmployee")
        IEmployee employeeObj;

        @GetMapping (path = "/fetchEmployee")
        public String fetchEmployee(){
            employeeObj.fetchEmployeeMethod();
            return "item fetched";
        }
    }

    @Aspect
    @Component
    public class LoggingAspect {

        @Before("target(com.conceptandcoding.learningspringboot.IEmployee)")
        public void beforeMethod() {
            System.out.println("inside beforeMethod aspect");
        }
    }

    public interface IEmployee {
        public void fetchEmployeeMethod();
    }

    @Component
    @Qualifier("tempEmployee")
    public class TempEmployee implements IEmployee{
        @Override
        public void fetchEmployeeMethod() {
            System.out.println("in temp Employee fetch method");
        }
    }

    @Component
    @Qualifier("permaEmployee")
    public class PermanentEmployee implements IEmployee{
        @Override
        public void fetchEmployeeMethod() {
            System.out.println("inside permanent fetch employee method");
        }
    }

    2024-06-22T12:12:42.485+05:30 INFO 67695 --- [           main] c.c.l.SpringbootApplication          : Started SpringbootApplication
    2024-06-22T12:12:48.192+05:30 INFO 67695 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[]   : Initializing Spring DispatcherServlet
    2024-06-22T12:12:48.192+05:30 INFO 67695 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet      : Initializing Servlet 'dispatcherServlet'
    2024-06-22T12:12:48.193+05:30 INFO 67695 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet      : Completed initialization in 1
    inside beforeMethod aspect
    in temp Employee fetch method
  
```

Combining two pointcuts using:

&& (boolean and)  
|| (boolean or)

```
@RestController  
@RequestMapping(value = "/api/")  
public class EmployeeController {  
  
    @Autowired  
    EmployeeUtil employeeUtil;  
  
    @GetMapping (path = "/fetchEmployee")  
    public String fetchEmployee(){  
        return "item fetched";  
    }  
}
```

```
@Aspect  
@Component  
public class LoggingAspect {  
  
    @Before("execution(* com.conceptandcoding.learningspringboot.EmployeeController.*())"  
           +  
           " && @within(org.springframework.web.bind.annotation.RestController)")  
    public void beforeAndMethod() {  
        System.out.println("inside beforeAndMethod aspect");  
    }  
  
    @Before("execution(* com.conceptandcoding.learningspringboot.EmployeeController.*())"  
           +  
           " || @within(org.springframework.stereotype.Component)")  
    public void beforeOrMethod() {  
        System.out.println("inside beforeOrMethod aspect");  
    }  
}
```

```
2024-06-22T13:02:39.698+05:30 INFO 70500 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[] : Initializing Spring DispatcherServlet 'dispatcher'  
2024-06-22T13:02:39.698+05:30 INFO 70500 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'  
2024-06-22T13:02:39.699+05:30 INFO 70500 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 1 ms  
inside beforeAndMethod aspect  
inside beforeOrMethod aspect
```

### Named Pointcuts

```
@RestController  
@RequestMapping(value = "/api/")  
public class EmployeeController {  
  
    @Autowired  
    EmployeeUtil employeeUtil;  
  
    @GetMapping (path = "/fetchEmployee")  
    public String fetchEmployee(){  
        return "item fetched";  
    }  
}
```

```
@Aspect  
@Component  
public class LoggingAspect {  
  
    @Pointcut("execution(* com.conceptandcoding.learningspringboot.EmployeeController.*())")  
    public void customPointcutName() {  
        //always stays empty  
    }  
  
    @Before("customPointcutName()")  
    public void beforeMethod() {  
        System.out.println("inside beforeMethod aspect");  
    }  
}
```

localhost:8080/api/fetchEmployee

item fetched

```
2024-06-22T13:07:37.855+05:30 WARN 70647 --- [main] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. This  
2024-06-22T13:07:37.254+05:30 INFO 70647 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8080 (http) with context path  
2024-06-22T13:07:37.260+05:30 INFO 70647 --- [main] c.c.l.SpringbootApplication : Started SpringbootApplication in 1.526 seconds (process completed in 1.526 ms)  
2024-06-22T13:07:41.084+05:30 INFO 70647 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[] : Initializing Spring DispatcherServlet 'dispatcher'  
2024-06-22T13:07:41.085+05:30 INFO 70647 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'  
2024-06-22T13:07:41.085+05:30 INFO 70647 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 0 ms  
inside beforeMethod aspect
```

### **Advice:**

Its an action, which is taken @Before or @After or @Around the method execution.

```
@Component
@Aspect
public class LoggingAspect {

    @Before("execution(public String com.conceptandcoding.learningspringboot.Employee.fetchEmployee())")
    public void beforeMethod(){
        System.out.println("inside beforeMethod Aspect");
    }
}
```

← Advice

@Before or @After: its simple and we have already seen

### **@Around:**

As the name says, it surrounds the method execution (before and after both).

```
@RestController
@RequestMapping(value = "/api/")
public class EmployeeController {

    @Autowired
    EmployeeUtil employeeUtil;

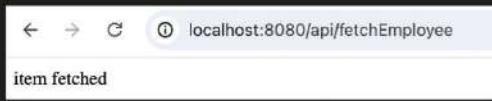
    @GetMapping (path = "/fetchEmployee")
    public String fetchEmployee(){
        return "item fetched";
    }
}

@Aspect
@Component
public class LoggingAspect {

    @Around("execution(* com.conceptandcoding.learningspringboot.EmployeeUtil.*())")
    public void aroundMethod(ProceedingJoinPoint joinPoint) throws Throwable {
        System.out.println("inside before Method aspect");
        joinPoint.proceed();
        System.out.println("inside after Method aspect");
    }
}

@Component
public class EmployeeUtil {

    public void employeeHelperMethod() {
        System.out.println("employee helper method called");
    }
}
```



```
2024-06-22T16:25:39.014+05:30 INFO 79523 --- [           main] c.c.l.SpringbootApplication      : Started SpringbootApplication in 1.011 seconds (JVM: 1.011s)
2024-06-22T16:25:41.031+05:30 INFO 79523 --- [nio-8080-exec-1] o.s.c.c.C.[Tomcat].[localhost].[/]   : Initializing Spring DispatcherServlet
2024-06-22T16:25:41.031+05:30 INFO 79523 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet    : Initializing Servlet 'dispatcherServlet'
2024-06-22T16:25:41.032+05:30 INFO 79523 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet    : Completed initialization in 1 ms
inside before Method aspect
fetching employee details
inside after Method aspect
```

**Join Point:** Its generally considered a point, where actual method invocation happens.

**By now, few questions we all should have:**

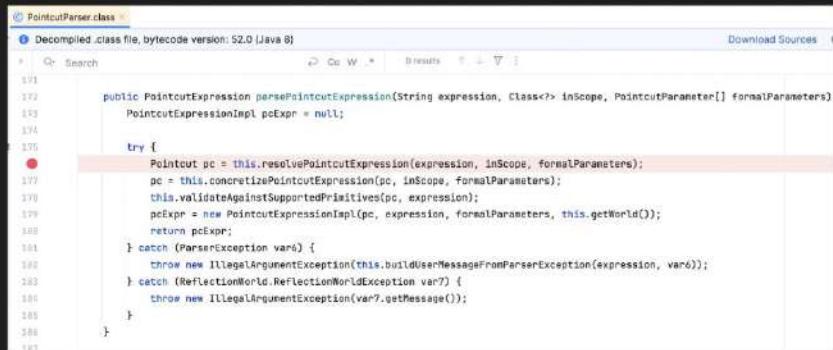
1. How this interception works?
2. What if we have 1000s of pointcut, so whenever I invoke a method, does matching happens with 1000s of pointcuts?

**Let's understand the AOP flow, to get an answer of above doubts:**

1. When Application startup happens
  - Look for @Aspect annotation Classes
  - Parse the Pointcut Expression
    - Done by *PointcutPaser.java* class
  - Stored in a efficient data structure or cache after parsing.
- Look for @Component, @Service @Controller etc.. annotation Classes
- For each class, it check if its eligible for interception based on pointcut expression
  - Done by *AbstractAutoProxyCreator.java* class
- If yes, it creates a Proxy using JDK Dynamic proxy or CGLIB proxy  
This proxy class, has code, which execute advice before the method, then method execution happens and after than advice if any.

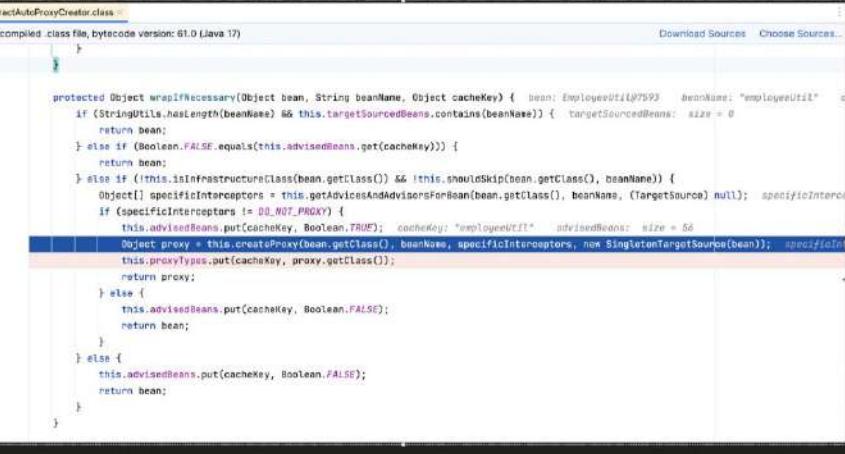
Now, when we initiate the Request after application startup, this Proxy class is actually invoked.

1. Parse the Pointcut Expression



```
171
172     public PointcutExpression parsePointcutExpression(String expression, Class<?> inScope, PointcutParameter[] formalParameters)
173         PointcutExpressionImpl pcExpr = null;
174
175     try {
176         Pointcut pc = this.resolvePointcutExpression(expression, inScope, formalParameters);
177         pc = this.concretizePointcutExpression(pc, inScope, formalParameters);
178         this.validateAgainstSupportedPrimitives(pc, expression);
179         pcExpr = new PointcutExpressionImpl(pc, expression, formalParameters, this.getWorld());
180         return pcExpr;
181     } catch (ParserException var6) {
182         throw new IllegalArgumentException(this.buildUserMessageFromParserException(expression, var6));
183     } catch (ReflectionWorld.ReflectionWorldException var7) {
184         throw new IllegalArgumentException(var7.getMessage());
185     }
186 }
187 }
```

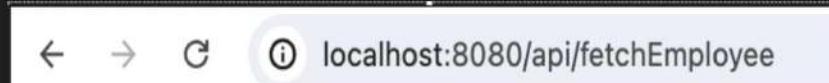
2. Create Proxy class (either JDK Dynamic proxy or CGLIB proxy) if required for the class



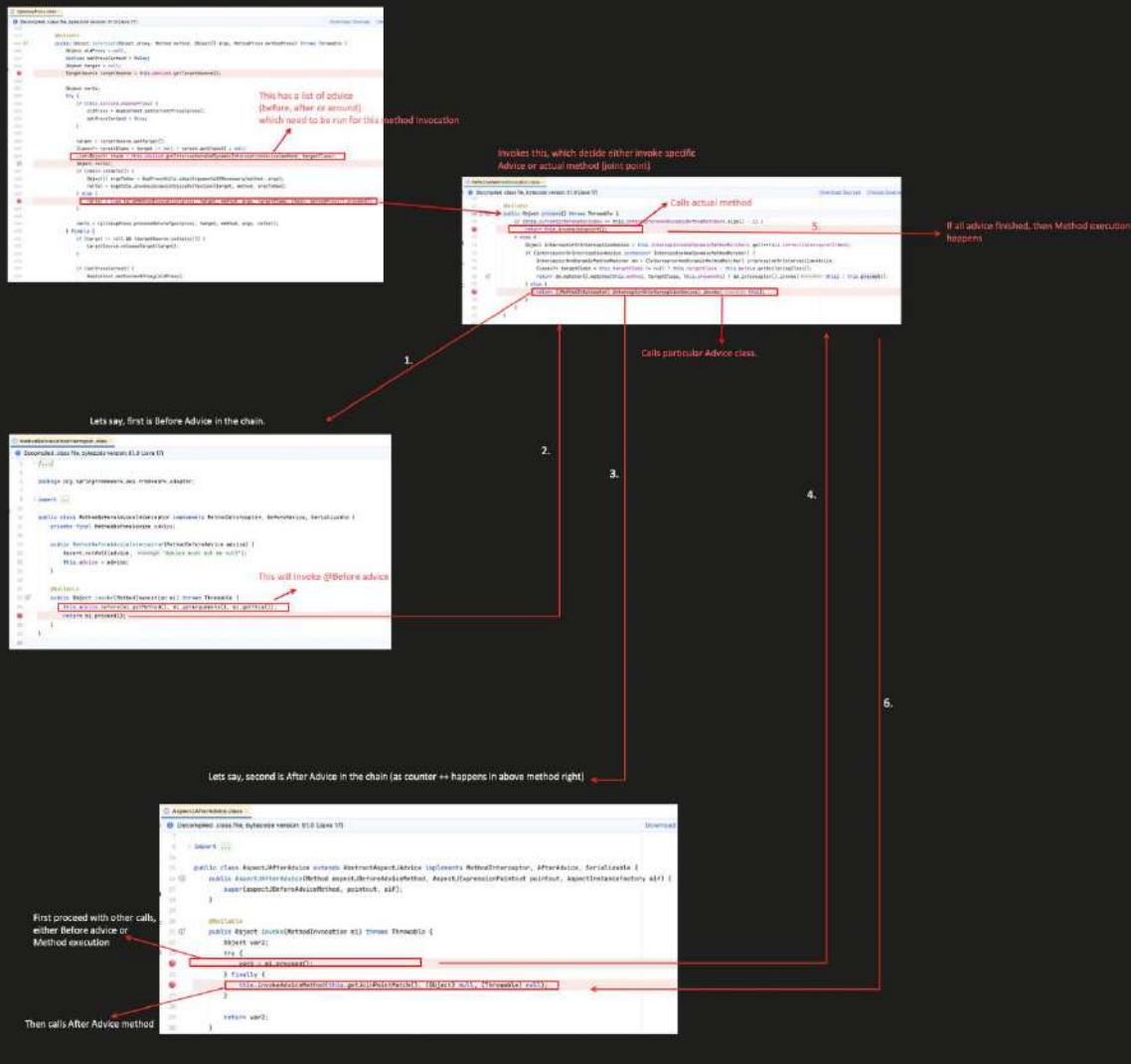
```
191
192
193
194     protected Object wrapIfNecessary(Object bean, String beanName, Object cacheKey) {    BeanNameUtil7593 beanNameUtil7593 = BeanNameUtil7593.get(beanName);    BeanNameUtil7593.set(beanNameUtil7593, beanName);
195     if (StringUtil.isNotEmpty(beanName) && this.targetSourcedBeans.contains(beanName)) {        targetSourcedBeans.size++;
196     return bean;
197     } else if (!Boolean.FALSE.equals(this.advisedBeans.get(cacheKey))) {
198         return bean;
199     } else if (!this.isInfrastructureClass(bean.getClass()) && !this.shouldSkip(bean.getClass(), beanName)) {
200         Object[] specificInterceptors = this.getAdvicesAndAdvisorsForBean(bean.getClass(), beanName, (TargetSource) null);
201         if (specificInterceptors != Boolean.FALSE) {
202             this.advisedBeans.put(cacheKey, Boolean.TRUE);    cacheKey: "employeeUtil"    advisedBeans: size = 58
203             Object proxy = this.createProxy(bean.getClass(), beanName, specificInterceptors, new SingletonTargetSource(bean));    specificInterceptors: size = 58
204             this.proxyTypes.put(cacheKey, proxy.getClass());
205             return proxy;
206         } else {
207             this.advisedBeans.put(cacheKey, Boolean.FALSE);
208             return bean;
209         }
210     } else {
211         this.advisedBeans.put(cacheKey, Boolean.FALSE);
212         return bean;
213     }
214 }
```

Application startup success

3. Invoke the request, which need Advice to be run.



4. Now, request actually tries to call Proxy Class, so it invokes either CGLIB proxy class or JDK proxy class.

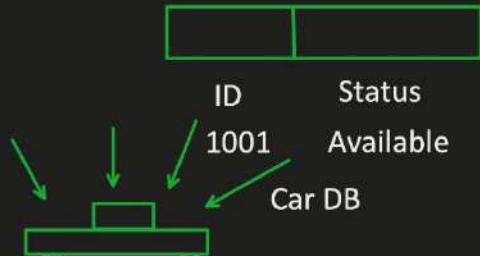


## Spring boot: @Transactional (Part1)

### Critical Section

Code segment, where shared resources are being accessed and modified.

```
{  
    Read Car Row with id: 1001  
    If Status is Available:  
        Update it to Booked  
}
```



When multiple request try to access this critical section, Data Inconsistency can happen.

Its solution is usage of **TRANSACTION**

- It helps to achieve ACID property.

#### A (Atomicity):

Ensures all operations within a transaction are completed successfully. If any operation fails, the entire transaction will get rollback.

#### C (Consistency):

Ensures that DB state before and after the transactions should be Consistent only.

#### I (Isolation):

Ensures that, even if multiple transactions are running in parallel, they do not interfere with each other.

#### Durability:

Ensures that committed transaction will never lost despite system failure or crash.

**BEGIN\_TRANSACTION:**

- Debit from A

- Credit to B

if all success:

**COMMIT;**

**Else**

**ROLLBACK;**

**END\_TRANSACTION;**

In Spring boot , we can use **@Transactional** annotation.

And for that:

1. we need to add below Dependency in pom.xml  
*(based on DB we are using, suppose we are using RELATIONAL DB)*

Spring boot Data JPA (Java persistence API): helps to interact with Relational databases without writing much code.

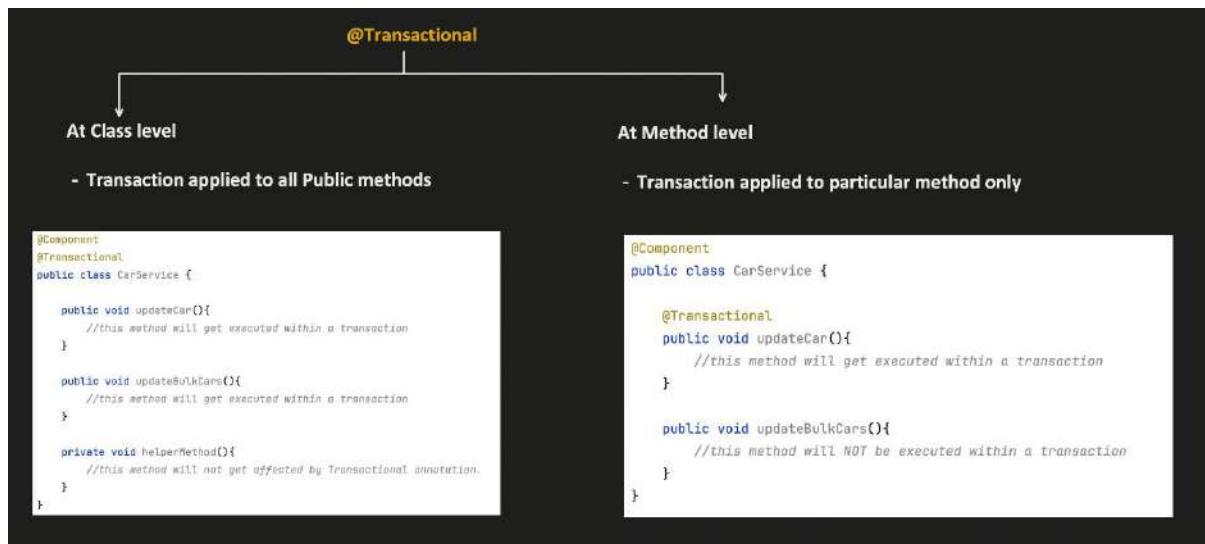
```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

+

Database driver dependency is also required (that we will see in next topic)

2. Activate Transaction Management by using `@EnableTransactionManagement` in main class.  
(spring boot generally Auto configure it, so we don't need to specially add it)

```
@SpringBootApplication  
@EnableTransactionManagement  
public class SpringbootApplication {  
  
    public static void main(String args[]) { SpringApplication.run(SpringbootApplication.class, args); }  
}
```



Transaction Management in Spring boot uses AOP.

- 1. Uses Point cut expression to search for method, which has @Transactional annotation like:  
`@within(org.springframework.transaction.annotation.Transactional)`

- 2. Once Point cut expression matches, run an "Around" type Advice.

Advice is:

`invokeWithinTransaction` method present present in `TransactionalInterceptor` class.

```
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @Autowired
    User user;

    @GetMapping(path = "/updateuser")
    public String updateUser(){
        user.updateUser();
        return "user is updated successfully";
    }
}
```

```
@Component
public class User {

    @Transactional
    public void updateUser(){
        System.out.println("UPDATE QUERY TO update the user db values");
    }
}
```

```

@Nullable
protected Object invokeWithinTransaction(Method method, @Nullable Class<?> targetClass,
                                         final InvocationCallback invocation) throws Throwable {
    :
    BEGIN_TRANSACTION
    try {
        TransactionInfo txInfo = createTransactionIfNecessary(txn, txAttr, joinpointIdentification);
        Object retVal;
        // This is un advised: Invoke the next interceptor in the chain.
        // This will normally result in a target object being invoked.
        retVal = invocation.proceedWithInvocation();
        :
        YOUR TASK
        :
        catch (Throwable ex) {
            // largest invocation exception
            completeTransactionAfterThrowing(txInfo, ex);
            throw ex;
        }
        finally {
            cleanupTransactionInfo(txInfo);
        }
        if (retVal != null && txAttr != null) {
            TransactionStatus status = txInfo.getTransactionStatus();
            if (status != null) {
                if (retVal instanceof Future<?> && future && future.isDone()) {
                    try {
                        future.get();
                    }
                    catch (ExecutionException ex) {
                        if (txAttr.isRollbackOnly(ex.getCause())) {
                            status.setRollbackOnly();
                        }
                    }
                    catch (InterruptedException ex) {
                        Thread.currentThread().interrupt();
                    }
                }
                else if (futurePresent && VavrDelegate.isVavrTry(retVal)) {
                    // Set rollback-only in case of Vavr failure violating our rollback rules...
                    retVal = VavrDelegate.evaluateTryFailure(retVal, txAttr, status);
                }
            }
        }
        :
        All success,
        COMMIT the txn
        :
        commitTransactionAfterReturning(txInfo);
        return retVal;
    }
}

```

Some more code here too in this method, but skipping them, just to avoid getting confused

**BEGIN\_TRANSACTION:**

- Debit from A
- Credit to B

If all success:

- COMMIT;
- Else
- ROLLBACK;

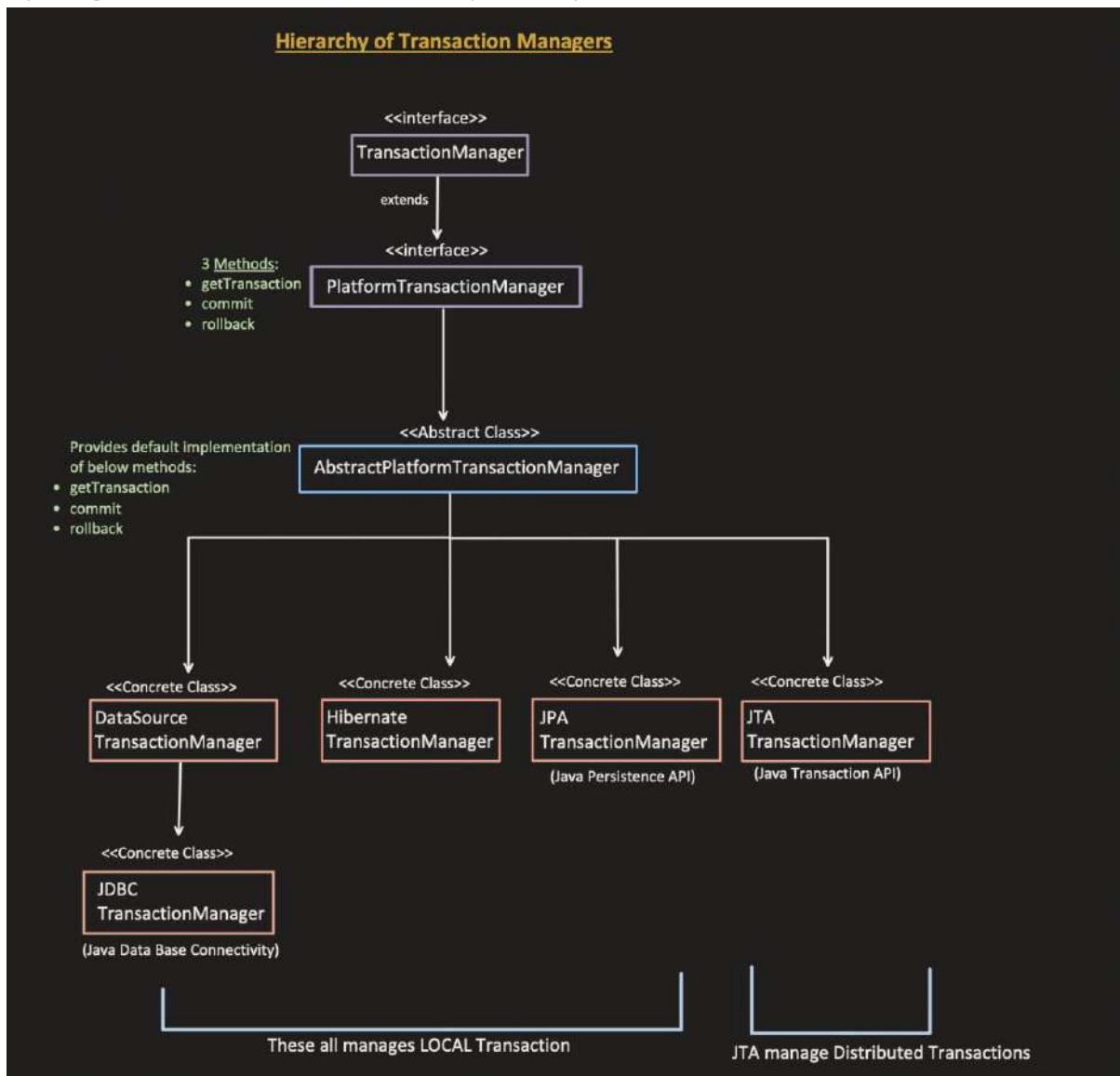
**END\_TRANSACTION;**

Some more code here too in this method, but skipping them, just to avoid getting confused

NOW, we know, how **TRANSACTIONAL** works, we will now see below topics in depth:

- **Transaction Context**
- **Transaction Manager**
  - Programmatic
  - Declarative
- **Propagation**
  - REQUIRED
  - REQUIRED\_NEW
  - SUPPORTS
  - NOT\_SUPPORTED
  - MANDATORY
  - NEVER
  - NESTED
- **Isolation level**
  - READ\_UNCOMMITTED
  - READ\_COMMITTED
  - REPEATABLE\_READ
  - SERIALIZABLE
- **Configure Transaction Timeout**
- **What is Read only transaction**
- etc..

## Spring boot: @Transactional (Part-2)



## Declarative

### Transaction Management through Annotation

```
@Component
public class User {

    @Transactional
    public void updateUser(){
        System.out.println("UPDATE QUERY TO update the user db values");
    }
}
```

Here, based on underlying DataSource used like JDBC or JPA etc.  
Spring boot will choose appropriate Transaction manager.

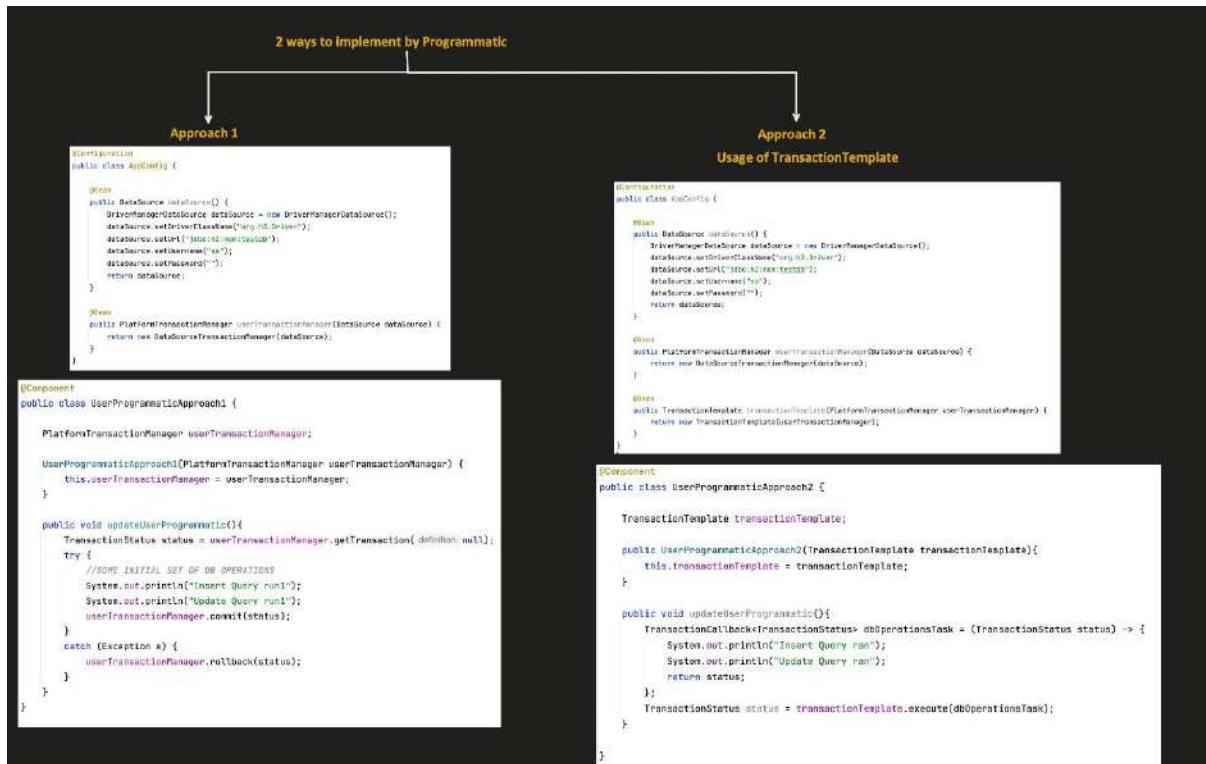
```
@Configuration
public class AppConfig {

    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("org.h2.Driver");
        dataSource.setUrl("jdbc:h2:mem:testdb");
        dataSource.setUsername("sa");
        dataSource.setPassword("");
        return dataSource;
    }

    @Bean
    public PlatformTransactionManager userTransactionManager(DataSource dataSource) {
        return new DataSourceTransactionManager(dataSource);
    }
}
```

```
@Component
public class UserDeclarative {

    @Transactional(transactionManager = "userTransactionManager")
    public void updateUserProgrammatic() {
        //SOME DB OPERATIONS
        System.out.println("Insert Query ran");
        System.out.println("Update Query ran");
    }
}
```



## Now, lets see Propagation

When we try to create a new Transaction, it first check the PROPAGATION value set, and this tell whether we have to create new transaction or not.

- REQUIRED (default propagation):

```
@Transactional(propagation=Propagation.REQUIRED)
if(parent txn present)
    Use it;
else
    Create new transaction;
```

- REQUIRED\_NEW:

```
@Transactional(propagation=Propagation.REQUIRED_NEW)
if(parent txn present)
    Suspend the parent txn;
    Create a new Txn and once finished;
    Resume the parent txn;
else
    Create new transaction and execute the method;
```

- SUPPORTS:

```
@Transactional(propagation=Propagation.SUPPORTS)
if(parent txn present)
    Use it;
Else
    Execute the method without any transaction;
```

- NOT\_SUPPORTED:

```
@Transactional(propagation=Propagation.NOT_SUPPORTED)
if(parent txn present)
    Suspend the parent txn;
    Execute the method without any transaction;
    Resume the parent txn;
else
    Execute the method without any transaction;
```

- MANDATORY:

```
@Transactional(propagation=Propagation.MANDATORY)
if(parent txn present)
    Use it;
Else
    Throw exception;
```

- NEVER:

```
@Transactional(propagation=Propagation.MANDATORY)
if(parent txn present)
    Throw exception;
Else
    Execute the method without any transaction;
```

#### Declarative way of usage:

```
@Component
public class UserDeclarative {
    @Autowired
    UserRepository userRepository;

    @Transactional
    public void updateUser() {
        System.out.println("Is transaction active: " + TransactionSynchronizationManager.isActualTransactionActive());
        System.out.println("Current transaction name: " + TransactionSynchronizationManager.getCurrentTransactionName());

        System.out.println("Some initial DB operation");
        userRepository.updateUserWithRequiredPropagation();
        System.out.println("Some final DB operation");
    }

    public void updateUserFromTransactionalManagement() {
        System.out.println("Is transaction active: " + TransactionSynchronizationManager.isActualTransactionActive());
        System.out.println("Current transaction name: " + TransactionSynchronizationManager.getCurrentTransactionName());
        System.out.println("Some initial DB operation");
        TransactionSynchronizationManager.setRequiredPropagation();
        System.out.println("Some final DB operation");
    }
}

@Component
public class UserDAO {
    @Transactional(propagation = Propagation.REQUIRED)
    /**
     * if(parent tm present)
     *   use it
     * else
     *   create new
     */
    public void dbOperationWithRequiredPropagation() {
        //EXECUTE DB QUERIES
        boolean isTransactionActive = TransactionSynchronizationManager.isActualTransactionActive();
        String currentTransactionName = TransactionSynchronizationManager.getCurrentTransactionName();
        System.out.println("*****");
        System.out.println("Propagation.REQUIRED: Is transaction active: " + isTransactionActive);
        System.out.println("Propagation.REQUIRED: Current transaction name: " + currentTransactionName);
        System.out.println("*****");
    }
}
```

#### Output:

```
Is transaction active: true
Current transaction name: com.conceptandcoding.learningspringboot.TransactionManagement.UserDeclarative.updateUser
Some initial DB operation
*****
Propagation.REQUIRED: Is parent transaction active: true
Propagation.REQUIRED: Current transaction name: com.conceptandcoding.learningspringboot.TransactionManagement.UserDeclarative.updateUser
*****
Some final DB operation

Is transaction active: false
Current transaction name: null
Some initial DB operation
*****
Propagation.REQUIRED: Is parent transaction active: true
Propagation.REQUIRED: Current transaction name: com.conceptandcoding.learningspringboot.TransactionManagement.UserDAO.dbOperationWithRequiredPropagation
*****
Some final DB operation
```

### Programmatic way of usage:

#### (Approach 1)

```
@Component
public class UserDeclarative {

    @Autowired
    UserDAO userDAOobj;

    @Transactional
    public void updateUser() {
        System.out.println("Is transaction active: " + TransactionSynchronizationManager.isActualTransactionActive());
        System.out.println("Current transaction name: " + TransactionSynchronizationManager.getCurrentTransactionName());

        System.out.println("Some initial DB operation");
        userDAOobj.dbOperationWithRequiredPropagationUsingProgrammaticApproach1();
        System.out.println("Some final DB operation");
    }

    public void updateUserFromNonTransactionalMethod() {
        System.out.println("Is transaction active: " + TransactionSynchronizationManager.isActualTransactionActive());
        System.out.println("Current transaction name: " + TransactionSynchronizationManager.getCurrentTransactionName());
        System.out.println("Some initial DB operation");
        userDAOobj.dbOperationWithRequiredPropagationUsingProgrammaticApproach1();
        System.out.println("Some final DB operation");
    }
}

@Component
public class UserDAO {

    PlatformTransactionManager userTransactionManager;

    UserDAO(PlatformTransactionManager userTransactionManager) {
        this.userTransactionManager = userTransactionManager;
    }

    /**
     * if(parent txn present)
     *      use it
     * else
     *      create new
     */
    public void dbOperationWithRequiredPropagationUsingProgrammaticApproach1(){
        DefaultTransactionDefinition transactionDefinition = new DefaultTransactionDefinition();
        transactionDefinition.setName("Testing REQUIRED propagation");
        transactionDefinition.setPropagationBehavior(TransactionDefinition.PROPAGATION_REQUIRED);
        TransactionStatus status = userTransactionManager.getTransaction(transactionDefinition);
        try {
            //EXECUTE operation
            System.out.println("*****");
            System.out.println("Propagation.REQUIRED: Is transaction active: " + TransactionSynchronizationManager.isActualTransactionActive());
            System.out.println("Propagation.REQUIRED: Current transaction name: " + TransactionSynchronizationManager.getCurrentTransactionName());
            System.out.println("*****");
            userTransactionManager.commit(status);
        }
        catch (Exception e) {
            userTransactionManager.rollback(status);
        }
    }
}
```

```

(Approach 2) : using TransactionTemplate

```

@Confidential
public class Application {

    @Resource
    public DataSource dataSource = new DriverManagerDataSource();
    dataSource.setDriverClassName("org.postgresql.Driver");
    dataSource.setUrl("jdbc:postgresql://localhost:5432/postgres");
    dataSource.setUsername("root");
    dataSource.setPassword("root");
    return dataSource;
}

@Event
public PlatformTransactionManager userInTransactionHandler(DataSource dataSource) {
    return new DatabasePlatformTransactionManager(dataSource);
}

@Delete
public TransactionTemplate transactionTemplate(PlatformTransactionManager userInTransactionManager) {
    TransactionTemplate transactionTemplate = new TransactionTemplate(userInTransactionManager);
    transactionTemplate.setPropagationBehavior(TransactionDefinition.REQUIRED);
    transactionTemplate.setIsolationLevel(TransactionTemplate.REQUIRED_ISOLATION);
    return transactionTemplate;
}
}

```



```

@Component
public class UserService {

    @Autowired
    UserDAO userDAO;

    @Transactional
    public void updateUser() {
        System.out.println("Is transaction active: " + TransactionSynchronizationManager.isActualTransactionActive());
        System.out.println("Current transaction name: " + TransactionSynchronizationManager.getCurrentTransactionName());

        System.out.println("Some initial DB operation");
        userDAO.updateUserWithRequiredPropagationUsingProgrammaticApproach2();
        System.out.println("Some final DB operation");
    }

    public void updateUserFromNonTransactionalMethod() {
        System.out.println("Is transaction active: " + TransactionSynchronizationManager.isActualTransactionActive());
        System.out.println("Current transaction name: " + TransactionSynchronizationManager.getCurrentTransactionName());
        System.out.println("Some initial DB operation");
        userDAO.updateUserWithRequiredPropagationUsingProgrammaticApproach2();
        System.out.println("Some final DB operation");
    }
}

```



```

import org.springframework.transaction.annotation.Transactional;
public class UserDAO {

    TransactionTemplate transactionTemplate;

    @Autowired(TransactionTemplate transactionTemplate)
    this.transactionTemplate = transactionTemplate;

    /**
     * If current tx is present
     * => use it
     * else
     * create new
     */
    public void updateUserWithRequiredPropagationUsingProgrammaticApproach2() {
        TransactionCallback<TransactionStatus> operations = (TransactionStatus status) -> {
            //None connection for this method
            System.out.println("*****");
            System.out.println("Propagation.REQUIRED: Is transaction active: " + TransactionSynchronizationManager.isActualTransactionActive());
            System.out.println("Propagation.REQUIRED: Current transaction name: " + TransactionSynchronizationManager.getCurrentTransactionName());
            System.out.println("*****");
            return status;
        };
        TransactionStatus status = transactionTemplate.execute(operations);
    }
}


```


```

**Output:**

```

Is transaction active: true
Current transaction name: com.conceptandcoding.learningspringboot.TransactionManagement.UserDeclarative.updateUser
Some initial DB operation
*****
Propagation.REQUIRED: Is transaction active: true
Propagation.REQUIRED: Current transaction name: com.conceptandcoding.learningspringboot.TransactionManagement.UserDeclarative.updateUser
*****
Some final DB operation

Is transaction active: false
Current transaction name: null
Some initial DB operation
*****
Propagation.REQUIRED: Is transaction active: true
Propagation.REQUIRED: Current transaction name: TRANSACTION TEMPLATE REQUIRED PROPAGATION
*****
Some final DB operation

```

## Spring boot- @Transaction Part3 (Isolation Level)

**Isolation Level**

**Isolation level:**  
It tells, how the changes made by one transaction are visible to other transactions running in parallel.

```
@Transactional(propagation = Propagation.REQUIRED, isolation = Isolation.READ_COMMITTED)
public void updateUser() {
    //some operations here
}
```

Isolation Level	Dirty Read Possible	Non-Repeatable Read Possible	Phantom Read Possible
READ_UNCOMMITTED	Yes	Yes	Yes
READ_COMMITTED	No	Yes	Yes
REPEATABLE_COMMITTED	No	No	Yes
SERIALIZABLE	No	No	No

↑ Concurrency High  
↓ Concurrency Low

Default isolation level, depends upon the DB which we are using.  
Like Most relational Databases uses READ\_COMMITTED as default isolation, but again it depends upon DB to DB.

**Dirty Read Problem**

Transaction A reads the un-committed data of other transaction.  
and  
If other transaction get ROLLED BACK, the un-committed data which is read by Transaction A is known as Dirty Read.

Time	Transaction A	Transaction B	DB Status
T1	BEGIN_TRANSACTION	BEGIN_TRANSACTION	Id: 123 Status: free
T2		Update Row id:123 Status = booked	Id: 123 Status: booked (Not Committed by Transaction B yet)
T3	Read Row id:123 (Got status = booked)		Id: 123 Status: booked (Not Committed by Transaction B yet)
T4		Rollback	Id: 123 Status: Free (Un-committed changes of Txn B got Rolled Back)

### Non-Repeatable Read Problem

If suppose Transaction A, reads the same row several times and there is a chance that it gets different value, then its known as Non-Repeatable Read problem.

Transaction A		DB
T1	BEGIN_TRANSACTION	ID: 1 Status: Free
T2	Read Row ID:1 (reads status: Free)	ID: 1 Status: Free
T3		ID: 1 Status: Booked
T4	Read Row ID:1 (reads status: Booked)	ID: 1 Status: Booked
T5	COMMIT	

### Phantom Read Problem

If suppose Transaction A, executes same query several times but there is a chance that rows returned are different. Then its known as Phantom Read problem.

Transaction A		DB
T1	BEGIN_TRANSACTION	ID: 1, Status: Free ID: 3, Status: Booked
T2	Read Row where ID>0 and ID<5 (reads 2 rows ID:1 and ID:3)	ID: 1, Status: Free ID: 3, Status: Booked
T3		ID: 1, Status: Free ID: 2, Status: Free ID: 3, Status: Booked
T4	Read Row where ID>0 and ID<5 (reads 3 rows ID:1, ID:2 and ID:3)	ID: 1, Status: Free ID: 2, Status: Free ID: 3, Status: Booked
T5	COMMIT	

### DB Locking Types

Locking make sure that, no other transaction update the locked rows.

Lock Type	Another Shared Lock	Another Exclusive Lock
Have Shared Lock	Yes	NO
Have Exclusive Lock	NO	NO

- Shared Lock (S) also known as READ LOCK.
- Exclusive Lock(X) also known as WRITE LOCK.

Let's see this table again:

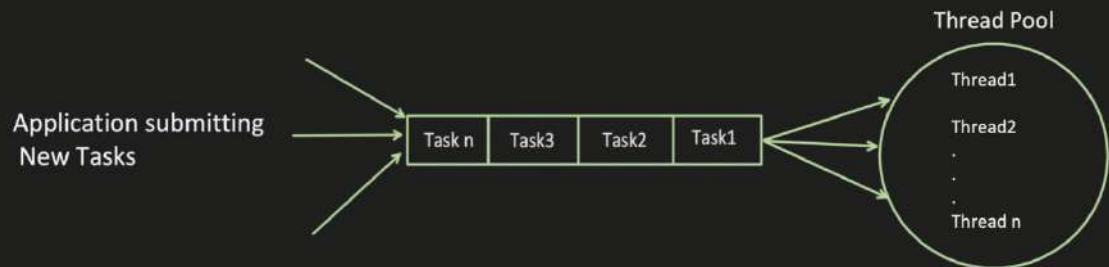
Isolation Level	Dirty Read Possible	Non-Repeatable Read Possible	Phantom Read Possible
READ_UNCOMMITTED	Yes	Yes	Yes
READ_COMMITTED	No	Yes	Yes
REPEATABLE_COMMITTED	No	No	Yes
SERIALIZABLE	No	No	No

ISOLATION LEVEL	Locking Strategy
<i>Read Uncommitted</i>	<b>Read :</b> No Lock acquired <b>Write :</b> No Lock acquired
<i>Read Committed</i>	<b>Read :</b> Shared Lock acquired and Released as soon as Read is done <b>Write :</b> Exclusive Lock acquired and keep till the end of the transaction
<i>Repeatable Read</i>	<b>Read :</b> Shared Lock acquired and Released only at the end of the Transaction <b>Write :</b> Exclusive Lock acquired and Released only at the end of the Transaction
<i>Serializable</i>	Same as Repeatable Read Locking Strategy + apply Range Lock and lock is released only at the end of the Transaction.

## Spring boot- Async Annotation (Part1)

### What is ThreadPool:

- It's a collection of threads (aka workers), which are available to perform the submitted tasks.
- Once task completed, worker thread get back to Thread Pool and wait for new task to assigned.
- Means threads can be reused.



In Java, thread pool is created using **ThreadPoolExecutor** Object

```
int minPoolSize = 2;
int maxPoolSize = 4;
int queueSize = 3;

ThreadPoolExecutor poolTaskExecutor = new ThreadPoolExecutor(minPoolSize, maxPoolSize, keepAliveTime: 1,
    TimeUnit.HOURS, new ArrayBlockingQueue<>(queueSize));
```

Now lets start understanding, how @Async Annotation works

### Async Annotation

- Used to mark method that should run asynchronously.
- Runs in a new thread, without blocking the main thread.

### Example:

```
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @Autowired
    UserService userServiceObj;

    @GetMapping(path = "/getuser")
    public String getUserMethod(){
        System.out.println("inside getUserMethod: " + Thread.currentThread().getName());
        userServiceObj.asyncMethodTest();
        return null;
    }
}

@Component
public class UserService {

    @Async
    public void asyncMethodTest() {
        System.out.println("inside asyncMethodTest: " + Thread.currentThread().getName());
    }
}
```

```
@SpringBootApplication
@EnableAsync
public class SpringbootApplication {

    public static void main(String args[]){
        SpringApplication.run(SpringbootApplication.class, args);
    }
}
```

### Output:

```
2024-08-11T15:27:21.985+05:30  INFO 44004 --- [nio-8080-exec-1]
2024-08-11T15:27:21.986+05:30  INFO 44004 --- [nio-8080-exec-1]
inside getUserMethod: http-nio-8080-exec-1
inside asyncMethodTest: task-1
inside getUserMethod: http-nio-8080-exec-2
inside asyncMethodTest: task-2
inside getUserMethod: http-nio-8080-exec-3
inside asyncMethodTest: task-3
inside getUserMethod: http-nio-8080-exec-4
inside asyncMethodTest: task-4
inside getUserMethod: http-nio-8080-exec-5
inside asyncMethodTest: task-5
inside getUserMethod: http-nio-8080-exec-6
inside asyncMethodTest: task-6
inside getUserMethod: http-nio-8080-exec-7
inside asyncMethodTest: task-7
inside getUserMethod: http-nio-8080-exec-8
inside asyncMethodTest: task-8
```

Creating new thread, each time we run

So, how does this "Async" Annotation, creates a new thread?

**Many places you will find, which says**

Spring boot uses by default "**SimpleAsyncTaskExecutor**", which creates new thread every time.

**I will say, this is not fully correct answer.**

**So, what's the right answer, What's the default Executor Spring boot uses?**

If we see below Spring boot framework code, it first looks for *defaultExecutor*, if no *defaultExecutor* found, only then *SimpleAsyncTaskExecutor* is used.

#### **AysncExecutionInterceptor.java**

```
@Nullable
protected Executor getDefaultExecutor(@Nullable BeanFactory beanFactory) {
    Executor defaultExecutor = super.getDefaultExecutor(beanFactory);
    return (Executor) (defaultExecutor != null ? defaultExecutor : new SimpleAsyncTaskExecutor());
}
```

### UseCase1:

```
@Configuration  
public class AppConfig {  
}  
  
@SpringBootApplication  
@EnableAsync  
public class SpringbootApplication {  
  
    public static void main(String args[]){  
        SpringApplication.run(SpringbootApplication.class, args);  
    }  
}  
  
@Component  
public class UserService {  
  
    @Async  
    public void asyncMethodTest() {  
        System.out.println("inside asyncMethodTest: " + Thread.currentThread().getName());  
    }  
}
```

During Application startup, Spring boot sees that, no **ThreadPoolTaskExecutor** Bean present, so it creates its default "**ThreadPoolTaskExecutor**" with below configurations.

```
defaultExecutor = {ThreadPoolTaskExecutor@7871}  
> poolSizeMonitor = {Object@7872}  
> corePoolSize = 8  
> maxPoolSize = 2147483647  
> keepAliveSeconds = 60  
> queueCapacity = 2147483647
```

**ThreadPoolTaskExecutor** is nothing but a Spring boot Object, which is just a wrapper around Java **ThreadPoolExecutor**.

*ThreadPoolTaskExecutor.java*

```
protected ExecutorService initializeExecutor(ThreadFactory threadFactory, RejectedExecutionHandler rejectedExecutionHandler) {  
    BlockingQueue<Runnable> queue = this.createQueue(this.queueCapacity);  
    ThreadPoolExecutor executor = new ThreadPoolExecutor(this.corePoolSize, this.maxPoolSize, (long) this.keepAliveSeconds, TimeUnit.SECONDS, queue, threadFactory, rejectedExecutionHandler) {  
        public void execute(Runnable command) {...}  
  
        protected void beforeExecute(Thread thread, Runnable task) {...}  
  
        protected void afterExecute(Runnable task, Throwable ex) {...}  
    };  
    if (this.allowCoreThreadTimeout) {...}  
    if (this.prestartAllCoreThreads) {...}  
    this.threadPoolExecutor = executor;  
    return executor;  
}
```

And its not recommended at all, why?

1. **Underutilization of Threads:** With Fixed Min pool size and Unbounded Queue(size is too big), its possible that most of the tasks will sit in the queue rather than creating new thread.
2. **High Latency:** Since queue size is too big, tasks will queue up till queue is not fill, high latency might occur during high load.
3. **Thread Exhaustion:** Lets say, if Queue also get filled up, then Executor will try to create new threads till Max pools size is not reached, which is Integer.MAX\_VALUE. This can lead to thread exhaustion. And server might go down because of overhead of managing so many threads.
4. **High Memory Usage:** Each threads need some memory, when we are creating these many threads, which may consume large amount of memory too, which might lead to performance degradation too.

#### UseCase2: Creating our own custom, **ThreadPoolTaskExecutor**

```
@Configuration  
public class AppConfig {  
  
    @Bean(name = "myThreadPoolExecutor")  
    public Executor taskPoolExecutor() {  
  
        int minPoolSize = 2;  
        int maxPoolSize = 4;  
        int queueSize = 3;  
  
        ThreadPoolTaskExecutor poolTaskExecutor = new ThreadPoolTaskExecutor();  
        poolTaskExecutor.setCorePoolSize(minPoolSize);  
        poolTaskExecutor.setMaxPoolSize(maxPoolSize);  
        poolTaskExecutor.setQueueCapacity(queueSize);  
        poolTaskExecutor.setThreadNamePrefix("MyThread-");  
        poolTaskExecutor.initialize();  
        return poolTaskExecutor;  
    }  
}
```

```
@Component  
public class UserService {  
  
    @Async  
    public void asyncMethodTest() {  
        System.out.println("Inside asyncMethodTest: " + Thread.currentThread().getName());  
    }  
}
```

```
@SpringBootApplication  
@EnableAsync  
public class SpringbootApplication {  
  
    public static void main(String args[]){  
        SpringApplication.run(SpringbootApplication.class, args);  
    }  
}
```

OR

```
@Component  
public class UserService {  
  
    @Async("myThreadPoolExecutor")  
    public void asyncMethodTest() {  
        System.out.println("Inside asyncMethodTest: " + Thread.currentThread().getName());  
    }  
}
```

During Application startup, Spring boot sees that, **ThreadPoolTaskExecutor** Bean present, so it makes it default only.

And even when we use **@Async** without any name, our "**myThreadPoolExecutor**" will get picked only.

### Output:

```
2024-08-11T17:05:14.403+05:30 INFO 47918 --- [nio-8080-exec-1] o.s.web.servlet.  
inside getUserMethod: http-nio-8080-exec-1  
inside asyncMethodTest: MyThread-1  
inside getUserMethod: http-nio-8080-exec-2  
inside asyncMethodTest: MyThread-2  
inside getUserMethod: http-nio-8080-exec-3  
inside asyncMethodTest: MyThread-1  
inside getUserMethod: http-nio-8080-exec-4  
inside asyncMethodTest: MyThread-2  
inside getUserMethod: http-nio-8080-exec-5  
inside asyncMethodTest: MyThread-1
```

### Output: (after putting sleep in async method, to simulate load)

```
@Component  
public class UserService {  
  
    @Async("myThreadPoolExecutor")  
    public void asyncMethodTest() {  
        System.out.println("inside asyncMethodTest: " + Thread.currentThread().getName());  
        try {  
            Thread.sleep( 5000 );  
        } catch ( Exception e ) {  
        }  
    }  
}
```

```
2024-08-11T20:43:02.142+05:30 INFO 49592 --- [nio-8080-exec-1]  
inside getUserMethod: http-nio-8080-exec-1  
inside asyncMethodTest: MyThread-1  
inside getUserMethod: http-nio-8080-exec-2  
inside asyncMethodTest: MyThread-2  
inside getUserMethod: http-nio-8080-exec-3  
inside getUserMethod: http-nio-8080-exec-4  
inside getUserMethod: http-nio-8080-exec-5  
inside getUserMethod: http-nio-8080-exec-6  
inside asyncMethodTest: MyThread-3  
inside getUserMethod: http-nio-8080-exec-7  
inside asyncMethodTest: MyThread-4  
inside getUserMethod: http-nio-8080-exec-8
```

Min Pool threads used

Queue got full

New Threads created, till Max Pool Capacity

Any New Request got Rejected

Its recommended, as its solve all the issues existing with the previous use case

#### UseCase3: Creating our own custom, **ThreadPoolExecutor (java one)**

```
@Configuration
public class AppConfig implements AsyncConfigurer {

    @Bean(name = "myThreadPoolExecutor")
    public Executor taskPoolExecutor() {
        int minPoolSize = 2;
        int maxPoolSize = 4;
        int queueSize = 5;

        ThreadPoolExecutor poolExecutor = new ThreadPoolExecutor(minPoolSize, maxPoolSize,
                TimeUnit.SECONDS, new ArrayBlockingQueue<Runnable>(queueSize), new CustomThreadFactory());
        return poolExecutor;
    }

    class CustomThreadFactory implements ThreadFactory {
        private final AtomicInteger threadId = new AtomicInteger(1);

        @Override
        public Thread newThread(Runnable r) {
            Thread thread = new Thread(r);
            thread.setName("MyThread-" + threadId.getAndIncrement());
            return thread;
        }
    }
}

@Component
public class UserService {

    @Async
    public void asyncMethodTest() {
        System.out.println("inside asyncMethodTest: " + Thread.currentThread().getName());
    }
}
```

```
@SpringBootApplication
@EnableAsync
public class SpringbootApplication {

    public static void main(String args[]){
        SpringApplication.run(SpringbootApplication.class, args);
    }
}
```

During Application startup, Spring boot sees that, **ThreadPoolExecutor (java one)** Bean is present, so it do not create its own default **ThreadPoolTaskExecutor (spring wrapper one)** instead it set the default executor is "**SimpleAsyncTaskExecutor**"

Now, if we run the above code, what we see.

```
2024-08-11T23:53:52.093+05:30 INFO 58643 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet
inside getUserMethod: http-nio-8080-exec-1
2024-08-11T23:53:52.124+05:30 INFO 58643 --- [nio-8080-exec-1] .s.a.AnnotationAsyncExecutionInterceptor
inside asyncMethodTest: SimpleAsyncTaskExecutor-1
inside getUserMethod: http-nio-8080-exec-2
inside asyncMethodTest: SimpleAsyncTaskExecutor-2
inside getUserMethod: http-nio-8080-exec-3
inside asyncMethodTest: SimpleAsyncTaskExecutor-3
inside getUserMethod: http-nio-8080-exec-4
inside asyncMethodTest: SimpleAsyncTaskExecutor-4
inside getUserMethod: http-nio-8080-exec-5
inside asyncMethodTest: SimpleAsyncTaskExecutor-5
inside getUserMethod: http-nio-8080-exec-6
inside asyncMethodTest: SimpleAsyncTaskExecutor-6
inside getUserMethod: http-nio-8080-exec-7
inside asyncMethodTest: SimpleAsyncTaskExecutor-7
inside getUserMethod: http-nio-8080-exec-8
inside asyncMethodTest: SimpleAsyncTaskExecutor-8
inside getUserMethod: http-nio-8080-exec-9
inside asyncMethodTest: SimpleAsyncTaskExecutor-9
```

And its not recommended at all to use "**SimpleAsyncTaskExecutor**", why?

It just creates new thread every time. So it may lead to

1. **Thread Exhaustion:** just blindly creating new thread with every Async request, might lead up to thread exhaustion.
2. **Thread Creation Overhead:** Since Threads are not reused, so thread management (creation, destroying) is an additional overhead.
3. **High Memory Usage:** Each threads need some memory, when we are creating these many threads, which may consume large amount of memory too, which might lead to performance degradation too.

So, whenever we have defined our own **ThreadPoolExecutor** (Java one), always specify the name also with **Async annotation**.

```
@Configuration
public class AppConfig implements AsyncConfigurer {

    @Bean(name = "myThreadPoolExecutor")
    public Executor taskPoolExecutor() {
        int minPoolSize = 2;
        int maxPoolSize = 4;
        int queueSize = 3;

        ThreadPoolExecutor poolExecutor = new ThreadPoolExecutor(minPoolSize, maxPoolSize,
                TimeUnit.SECONDS, new ArrayBlockingQueue<Runnable>(queueSize), new CustomThreadFactory());
        poolExecutor.setKeepAliveTime(1, TimeUnit.HOURS);
        return poolExecutor;
    }

    class CustomThreadFactory implements ThreadFactory {
        private final AtomicInteger threadNo = new AtomicInteger( initialValue: 1);

        @Override
        public Thread newThread(Runnable r) {
            Thread thread = new Thread(r);
            thread.setName("MyThread-" + threadNo.getAndIncrement());
            return thread;
        }
    }
}

@Component
public class UserService {

    @Async("myThreadPoolExecutor")
    public void asyncMethodTest() {
        System.out.println("inside asyncMethodTest: " + Thread.currentThread().getName());
    }
}
```

#### Output:

```
2024-08-11T17:05:14.403+05:30  INFO 47918 --- [nio-8080-exec-1] o.s.web.servlet.
inside getUserMethod: http-nio-8080-exec-1
inside asyncMethodTest: MyThread-1
inside getUserMethod: http-nio-8080-exec-2
inside asyncMethodTest: MyThread-2
inside getUserMethod: http-nio-8080-exec-3
inside asyncMethodTest: MyThread-1
inside getUserMethod: http-nio-8080-exec-4
inside asyncMethodTest: MyThread-2
inside getUserMethod: http-nio-8080-exec-5
inside asyncMethodTest: MyThread-1
```

Hey, I don't want all this confusion, Usecase1, Usecase2 or Usecase3.

I always want to set my executor as default one, even if anyone use @Async, my executor only should be picked.

```
@Configuration
public class AppConfig implements AsyncConfigurer {

    private ThreadPoolExecutor poolExecutor;

    @Override
    public Executor getAsyncExecutor() {
        Executor executor = null;
        if (poolExecutor == null) {
            int minPoolSize = 2;
            int maxPoolSize = 4;
            int queueSize = 3;
            poolExecutor = new ThreadPoolExecutor(minPoolSize, maxPoolSize, 1000L, TimeUnit.MILLISECONDS, new ArrayBlockingQueue<Runnable>(queueSize), new CustomThreadFactory());
        }
        return poolExecutor;
    }

    class CustomThreadFactory implements ThreadFactory {
        private final AtomicInteger threadNumber = new AtomicInteger(1);

        @Override
        public Thread newThread(Runnable r) {
            Thread th = new Thread(r);
            th.setName("MyThread-" + threadNumber.getAndIncrement());
            return th;
        }
    }
}
```

### Using Java ThreadPoolExecutor

```
@Component
public class UserService {

    @Async
    public void asyncMethodTest() {
        System.out.println("inside asyncMethodTest: " + Thread.currentThread().getName());
    }
}
```

No, Executor name provided

Still, default executor configuration picked is mine one, not SimpleAsyncTaskExecutor

Output:

```
2024-08-12T08:11:42.878+05:30  INFO 58958 --- [nio-8080-exec-1]
inside getUserMethod: http-nio-8080-exec-1
inside asyncMethodTest: MyThread-1
inside getUserMethod: http-nio-8080-exec-2
inside asyncMethodTest: MyThread-2
inside getUserMethod: http-nio-8080-exec-3
inside asyncMethodTest: MyThread-3
inside getUserMethod: http-nio-8080-exec-4
inside asyncMethodTest: MyThread-4
inside getUserMethod: http-nio-8080-exec-5
inside asyncMethodTest: MyThread-5
```

## SpringBoot @Async Annotation- Part2

### Conditions for @Async Annotation to work properly?

#### 1. Different Class :

If @Async annotation is applied to the method within the same class from which it is being called, then Proxy mechanism is skipped because internal method calls are **NOT INTERCEPTED**.

#### 2. Public method:

Method annotated with @Async must be public. And again, AOP interception works only on Public methods.

```
RestController  
@RequestMapping(value = "/api/")  
public class UserController {  
  
    @GetMapping(path = "/getuser")  
    public String getUserMethod(){  
        System.out.println("inside getUserMethod: " + Thread.currentThread().getName());  
        asyncMethodTest();  
        return null;  
    }  
  
    @Async  
    public void asyncMethodTest() {  
        System.out.println("inside asyncMethodTest: " + Thread.currentThread().getName());  
    }  
}
```

 This is wrong way of doing it, correct way is,  
@Async method should be in different class and  
should be public

### Output:

```
2024-08-18T12:46:04.532+05:30  INFO 60618 ---  
inside getUserMethod: http-nio-8080-exec-1  
inside asyncMethodTest: http-nio-8080-exec-1
```

## @Aysn and Transaction Management

### Usecase1:



Transaction Context do not transfer from caller thread to new thread which got created by Async.

```
@RestController  
@RequestMapping(value = "/api/")  
public class UserController {  
  
    @Autowired  
    UserService userService;  
  
    @PostMapping(path = "/updateuser")  
    public String updateUserMethod(){  
        userService.updateUser();  
        return null;  
    }  
}
```

```
@Component  
public class UserService {  
  
    @Autowired  
    UserUtility userUtility;  
  
    @Transactional  
    public void updateUser(){  
  
        //1. update user status  
        //2. update user first name  
  
        //3. update user  
        userUtility.updateUserBalance();  
    }  
}
```

```
@Component  
public class UserUtility {  
  
    @Async  
    public void updateUserBalance(){  
        //updating user balance amount.  
    }  
}
```

**Usecase2:** Use with Precaution, as new thread will be created and have transaction management too but context is not same as parent thread. So Propagation will not work as expected.

```
@RestController  
@RequestMapping(value = "/api/")  
public class UserController {  
  
    @Autowired  
    UserService userService;  
  
    @PostMapping(path = "/updateuser")  
    public String updateUserMethod(){  
        userService.updateUser();  
        return null;  
    }  
}
```

```
@Component  
public class UserService {  
  
    @Transactional  
    @Async  
    public void updateUser(){  
  
        //1. update user status  
        //2. update user first name  
        //3. update user  
    }  
}
```

**Usecase3:** ✓

```
@RestController  
@RequestMapping(value = "/api/")  
public class UserController {  
  
    @Autowired  
    UserService userService;  
  
    @PostMapping(path = "/updateuser")  
    public String updateUserMethod(){  
        userService.updateUser();  
        return null;  
    }  
}
```

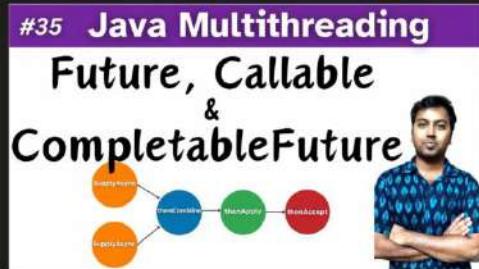
```
@Component  
public class UserService {  
  
    @Autowired  
    UserUtility userUtility;  
  
    @Async  
    public void updateUser(){  
        userUtility.updateUser();  
    }  
}
```

```
@Component  
public class UserUtility {  
  
    @Transactional  
    public void updateUser(){  
        //1. update user status  
        //2. update user first name  
        //3. update user  
    }  
}
```

### @Aysn Method return type

Both Future and Completable Future can be the return type of the Async method

*Checkout Java Playlist to learn more in depth of Future and CompletableFuture*



#### **Using Future:**

```
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @Autowired
    UserService userService;

    @GetMapping(path = "/getuser")
    public String getUserMethod(){
        Future<String> result = userService.performTaskAsync();
        String output = null;
        try {
            output = result.get();
            System.out.println(output);
        }catch (Exception e) {
            System.out.println("some exception");
        }
        return output;
    }
}
```

```
@Component
public class UserService {

    @Async
    public Future<String> performTaskAsync(){
        return new AsyncResult<>(value: "async task result");
    }
}
```

S.No.	Method Available in Future Interface	Purpose
1.	<code>boolean cancel(boolean mayInterruptIfRunning)</code>	<ul style="list-style-type: none"> <li>Attempts to cancel the execution of the task.</li> <li>Returns false, if task can not be cancelled (typically bcoz task already completed); returns true otherwise.</li> </ul>
2.	<code>boolean isCancelled()</code>	<ul style="list-style-type: none"> <li>Returns true, if task was cancelled before it get completed.</li> </ul>
3.	<code>boolean isDone()</code>	<ul style="list-style-type: none"> <li>Returns true if this task completed. Completion may be due to normal termination, an exception, or cancellation -- in all of these cases, this method will return true.</li> </ul>
4.	<code>V get()</code>	<ul style="list-style-type: none"> <li>Wait if required, for the completion of the task.</li> <li>After task completed, retrieve the result if available.</li> </ul>
5.	<code>V get(long timeout, TimeUnit unit)</code>	<ul style="list-style-type: none"> <li>Wait if required, for at most the given timeout period.</li> <li>Throws 'TimeoutException' if timeout period finished and task is not yet completed.</li> </ul>

### Using CompletableFuture:

Introduced in Java8

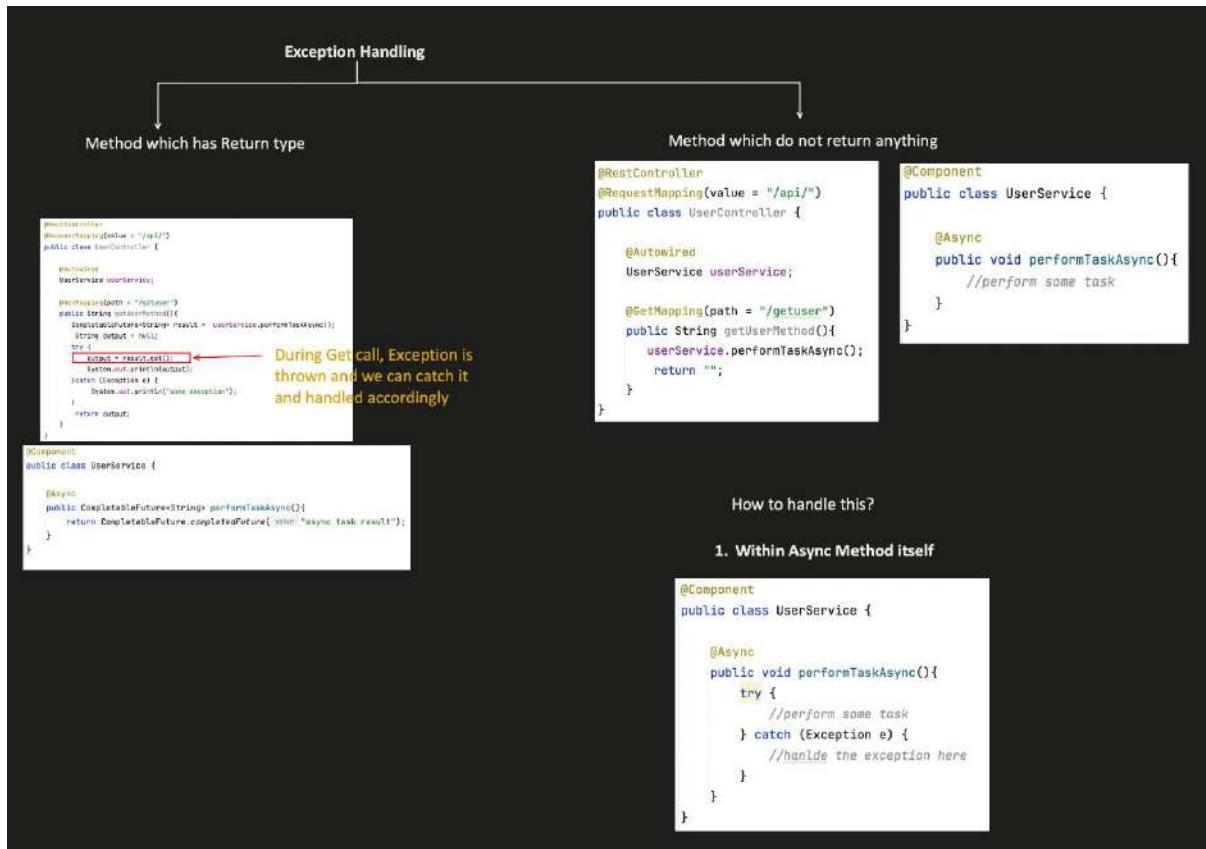
```
@RestController
@RequestMapping(value = "/api/*")
public class UserController {

    @Autowired
    UserService userService;

    @GetMapping(path = "/getuser")
    public String getUserMethod(){
        CompletableFuture<String> result = userService.performTaskAsync();
        String output = null;
        try {
            output = result.get();
            System.out.println(output);
        }catch (Exception e) {
            System.out.println("some exception");
        }
        return output;
    }
}
```

```
@Component
public class UserService {

    @Async
    public CompletableFuture<String> performTaskAsync(){
        return CompletableFuture.completedFuture( value: "async task result");
    }
}
```



## 2. Implement Custom AsyncExceptionHandler

```
@Configuration
public class AppConfig implements AsyncConfigurer {

    @Autowired
    private AsyncUncaughtExceptionHandler asyncUncaughtExceptionHandler;

    @Override
    public AsyncUncaughtExceptionHandler getAsyncUncaughtExceptionHandler() {
        return this.asyncUncaughtExceptionHandler;
    }

    @Component
    class DefaultAsyncUncaughtExceptionHandler implements AsyncUncaughtExceptionHandler {

        @Override
        public void handleUncaughtException(Throwable ex, Method method, Object... params) {
            System.out.println("in default Uncaught Exception method");
            //logging can be done here.
        }
    }
}
```

```
@Component
public class UserService {

    @Async
    public void performTaskAsync(){
        int i = 0;
        System.out.println(5/i);
    }
}
```

### Output:

```
2024-08-18T20:15:14.082+05:30  INFO 70455 --- [nio-8080-exec-1]
in default Uncaught Exception method
```

If , we will not handle it, then, Spring boot default SimpleAsyncUncaughtExceptionHandler will get invoked

```
@Configuration
public class AppConfig {
}
```

```
@Component
public class UserService {

    @Async
    public void performTaskAsync(){
        int i = 0;
        System.out.println(5/i);
    }
}
```

### Spring boot framework code..

```
public class SimpleAsyncUncaughtExceptionHandler implements AsyncUncaughtExceptionHandler {  
    private static final Log logger = LoggerFactory.getLog(SimpleAsyncUncaughtExceptionHandler.class);  
  
    public SimpleAsyncUncaughtExceptionHandler() {}  
  
    public void handleUncaughtException(Throwable ex, Method method, Object... params) {  
        if (logger.isErrorEnabled()) {  
            logger.error("Unexpected exception occurred invoking async method: " + method, ex);  
        }  
    }  
}
```

### Output:

```
task-5] .a.i.SimpleAsyncUncaughtExceptionHandler : Unexpected exception occurred invoking async method  
java.lang.ArithmetricException Create breakpoint : / by zero  
at com.conceptandcoding.learningspringboot.AsyncAnnotationLearn.UserService.performTaskAsync(UserService.java:18)
```

## Springboot: Custom Interceptors

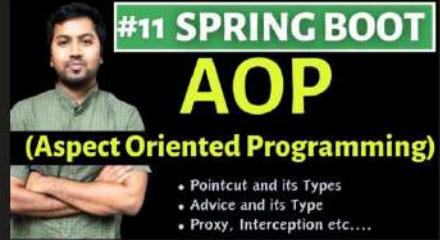
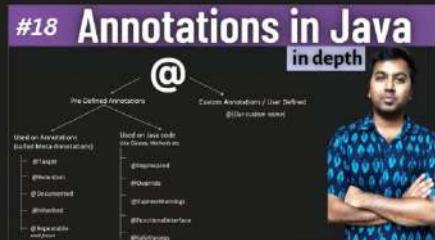
### Interceptor:

it's a mediator, which get invoked before or after your actual code.

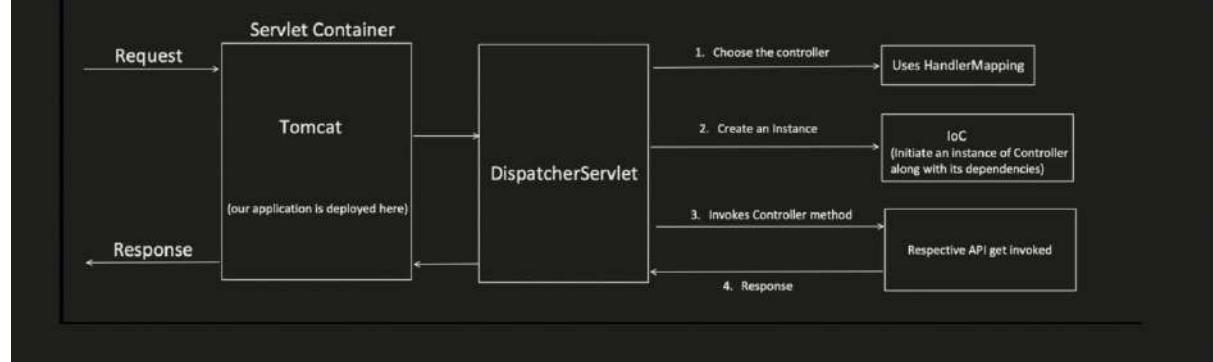
In future topics below, we might need to write our custom interceptors:

- *Springboot Caching,*
- *Springboot logging,*
- *Springboot Authentication etc..*

Pre-requisite to understand custom interceptor better:



Custom Interceptor for Requests before even reaching to specific Controller class



```

@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @Autowired
    User user;

    @GetMapping(path = "/getUser")
    public String getUser() {
        user.getUser();
        return "Success";
    }
}

@Configuration
public class AppConfig implements WebMvcConfigurer {

    @Autowired
    MyCustomInterceptor myCustomInterceptor;

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(myCustomInterceptor)
            .addPathPatterns("/api/*") // Apply to these URL patterns
            .excludePathPatterns("/api/updateUser", "/api/deleteUser"); // Exclude for these URL patterns
    }
}

```

```

http://localhost:8080/api/getUser

2024-08-31T23:01:49.217+05:30  INFO 4417 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet      : Completed initialization in 0 ms
inside pre handle Method
hitting db to get the userdata
inside post handle method
inside after completion method

```

## Custom Interceptor for Requests after reaching to specific Controller class

### Step1: Creation of custom annotation

We can create Custom Annotation using keyword "`@interface`" java Annotation.

```
public @interface MyCustomAnnotation {  
}  
  
public class User {  
  
    @MyCustomAnnotation  
    public void updateUser(){  
        //some business logic  
    }  
}
```

2 Important Meta Annotation properties are:

- **@Target :**

this meta annotation, tells where we can apply the particular annotation on method or class or constructor etc.

```
@Target(ElementType.METHOD)
public @interface MyCustomAnnotation {
}
```

```
@Target({ElementType.CONSTRUCTOR, ElementType.METHOD, ElementType.PARAMETER, ElementType.FIELD})
public @interface MyCustomAnnotation {
}
```

**@Retention:**

this meta annotation tell, how the particular annotation will be stored in java.

▪ **RetentionPolicy.SOURCE :**

*Annotation will be discarded by compiler itself and its not even recorded in .class file.*

**RetentionPolicy.CLASS :**

*Annotation will be recorded in .class file but ignored by JVM during run time.*

**RetentionPolicy.RUNTIME :**

*Annotation will be recorded in .class file and also available during run time.*

The screenshot displays two side-by-side code editors and their corresponding decompiled bytecode windows. The left editor shows Java code with `@Retention(RetentionPolicy.SOURCE)`. The right editor shows Java code with `@Retention(RetentionPolicy.RUNTIME)`. Arrows point from each editor to its respective decompiled bytecode window above it.

**Left Editor (RetentionPolicy.SOURCE):**

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface MyCustomAnnotation {
}

public class User {
    @MyCustomAnnotation
    public void updateUser(){
        //some business logic
    }
}
```

**Right Editor (RetentionPolicy.RUNTIME):**

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface MyCustomAnnotation {
}

public class User {
    @MyCustomAnnotation
    public void updateUser(){
        //some business logic
    }
}
```

**Decompiled Bytecode (Left):**

```
Decomplied class file, bytecode version: 51.0 (Java 17)

public class User {
    public void updateUser();
}
```

**Decompiled Bytecode (Right):**

```
Decompled class file, bytecode version: 51.0 (Java 17)

package com.conceptandcoding.learningspringboot.CustomAnnotationTesting;

public class User {
    public void updateUser();
}
```

How to create Custom Annotation with methods (more like a fields):

- No parameter, no body
- Return type is restricted to:
  - Primitive type (int, boolean, double etc. )
  - String
  - Enum
  - Class<?>
  - Annotations
  - Array of above types

```
public class User {  
  
    @Target(ElementType.METHOD)  
    @Retention(RetentionPolicy.RUNTIME)  
    public @interface MyCustomAnnotation {  
  
        String key() default "defaultKeyName";  
    }  
}  
  
  
@Target(ElementType.METHOD)  
@Retention(RetentionPolicy.RUNTIME)  
public @interface MyCustomAnnotation {  
  
    int intKey() default 0;  
  
    String stringKey() default "defaultString";  
  
    Class<?> classTypeKey() default String.class;  
  
    MyCustomEnum enumKey() default MyCustomEnum.ENUM_VAL1;  
  
    String[] stringArrayKey() default {"default1", "default2"};  
  
    int[] intArrayKey() default {1, 2};  
}
```

### Step2: Creation of Custom Interceptor

```
@Target({ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface MyCustomAnnotation {
    String name() default "";
}

@RestController
@RequestMapping(value = "/api/")
public class UserController {
    @Autowired
    User user;

    @GetMapping(path = "/getUser")
    public String getUser(){
        user.getUser();
        return "success";
    }
}

@Component
public class User {
    @MyCustomAnnotation(name = "user")
    public void getUser() {
        System.out.println("get the user details");
    }
}

@Component
@Aspect
public class MyCustomInterceptor {

    @Around("@annotation(com.conceptandcoding.learningspringboot.CustomInterceptor.MyCustomAnnotation)") //pointcut expression
    public void invoke(ProceedingJoinPoint joinPoint) throws Throwable{ //advice

        System.out.println("do something before actual method");

        Method method = ((MethodSignature)joinPoint.getSignature()).getMethod();
        if(method.isAnnotationPresent(MyCustomAnnotation.class)){
            MyCustomAnnotation annotation = method.getAnnotation(MyCustomAnnotation.class);
            System.out.println("name from annotation: " + annotation.name());
        }

        joinPoint.proceed();
        System.out.println("do something after actual method");
    }
}
```

### Output:

```
2024-08-31T17:51:51.344+05:30 INFO 3464 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet      : Completed initialization in 0 ms
do something before actual method
name from annotation: user
get the user details
do something after actual method
```

## Springboot: Filters Vs Interceptors

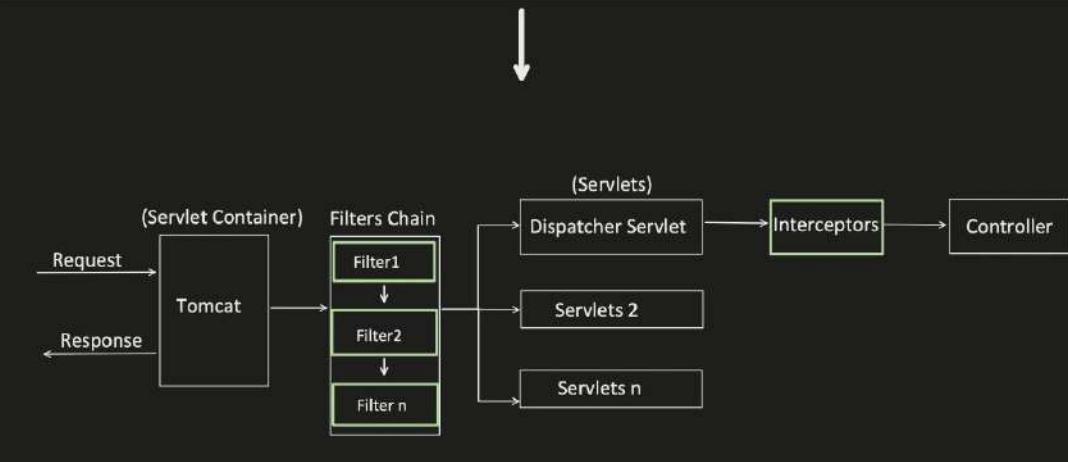
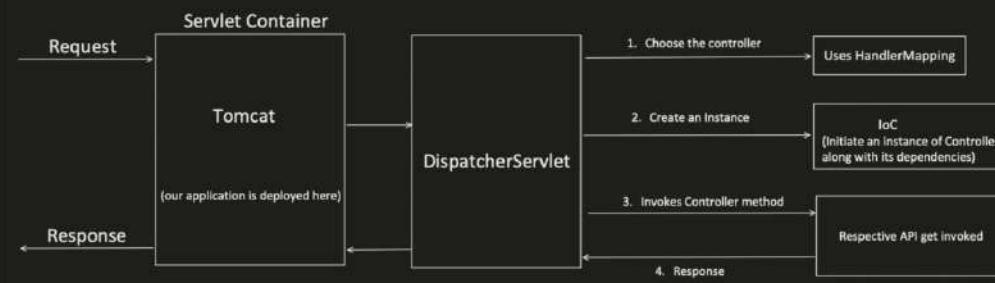
### Filter:

It intercepts the HTTP Request and Response, before they reach to the servlet.

### Interceptor:

It's specific to Spring framework, and intercepts HTTP Request and Response, before they reach to the Controller.

Lets see this diagram again:



**What is servlet:**

Servlet is nothing but a Java class, which accepts the incoming request, process it and returns the response.

We can create multiple servlets like:

Servlet 1: can be configured to handle REST APIs

Servlet 2 : can be configured to handle SOAP APIs etc....

Similarly like this, "**DispatcherServlet**" is kind of servlet provided by spring, and by default its configured to handle all APIs "/\*".

**Filter:**

Is used when we want to intercept HTTP Request and Response and add logic agnostic of the underlying servlets.

We can have many filters and have ordering between them too.

**Interceptors:**

Is used when we want to intercept HTTP request and response and add logic specific to a particular servlet.

We can have many Interceptors and have ordering between them too.

### Multiple Interceptors and its Ordering:

In previous video, we already saw, how to add 1 interceptor.  
How "preHandle", "postHandle" and "afterCompletion" comes into the picture.

Now here, will show how to add more than 1.

Also, if "preHandle" returns false, next interceptor and controller will not get invoked itself.

```
@Configuration
public class AppConfig implements WebMvcConfigurer {

    @Autowired
    MyCustomInterceptor1 myCustomInterceptor1;

    @Autowired
    MyCustomInterceptor2 myCustomInterceptor2;

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        //below order is maintained while calling interceptors
        registry.addInterceptor(myCustomInterceptor1)
            .addPathPatterns("/api/*") // Apply to these URL patterns
            .excludePathPatterns("/api/updateUser", "/api/deleteUser"); // Exclude for these URL patterns

        registry.addInterceptor(myCustomInterceptor2)
            .addPathPatterns("/api/*") // Apply to these URL patterns
            .excludePathPatterns("/api/updateUser");
    }
}
```

### Output:

---

```
2024-09-02T17:01:11.719+05:30  INFO 7162 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet
2024-09-02T17:01:11.720+05:30  INFO 7162 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet
inside pre handle Method - MyCustomInterceptor1
inside pre handle Method - MyCustomInterceptor2
hitting db to get the userdata
inside post handle method- MyCustomInterceptor2
inside post handle method- MyCustomInterceptor1
inside after completion method - MyCustomInterceptor2
inside after completion method- MyCustomInterceptor1
|
```

```

How to Add Filters:

import jakarta.servlet.*;
import java.io.IOException;

public class MyFilter1 implements Filter {
    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
        Filter.super.init(filterConfig);
    }

    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain)
            throws IOException, ServletException {
        System.out.println("MyFilter1 inside");
        filterChain.doFilter(servletRequest, servletResponse);
        System.out.println("MyFilter1 completed");
    }

    @Override
    public void destroy() {
        Filter.super.destroy();
    }
}

@Configuration
public class AppConfig {
    @Bean
    public FilterRegistrationBean<MyFilter1> myFilter1Registration() {
        FilterRegistrationBean<MyFilter1> filterRegistrationBean = new FilterRegistrationBean<MyFilter1>();
        filterRegistrationBean.setFilter(new MyFilter1());
        filterRegistrationBean.addPattern("/*");
        filterRegistrationBean.setName("MyFilter1");
        return filterRegistrationBean;
    }

    @Bean
    public FilterRegistrationBean<MyFilter2> myFilter2Registration() {
        FilterRegistrationBean<MyFilter2> filterRegistrationBean = new FilterRegistrationBean<MyFilter2>();
        filterRegistrationBean.setFilter(new MyFilter2());
        filterRegistrationBean.addPattern("/*");
        filterRegistrationBean.setName("MyFilter2");
        return filterRegistrationBean;
    }
}

```

**Output:**

-----

```

2024-09-02T17:30:39.279+05:30  INFO 7350 --- [nio-8080-exec-1]
2024-09-02T17:30:39.280+05:30  INFO 7350 --- [nio-8080-exec-1]
MyFilter2 inside
MyFilter1 inside
hitting db to get the userdata
MyFilter1 completed
MyFilter2 completed

```

If both Interceptor and Filter used together, Output will look like this.

```

2024-09-02T17:32:25.643+05:30  INFO 7373 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet
2024-09-02T17:32:25.644+05:30  INFO 7373 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet
MyFilter1 inside
MyFilter2 inside
inside pre handle Method - MyCustomInterceptor1
inside pre handle Method - MyCustomInterceptor2
hitting db to get the userdata
inside post handle method- MyCustomInterceptor2
inside post handle method- MyCustomInterceptor1
inside after completion method - MyCustomInterceptor2
inside after completion method- MyCustomInterceptor1
MyFilter2 completed
MyFilter1 completed

```

## Springboot: Filters Vs Interceptors

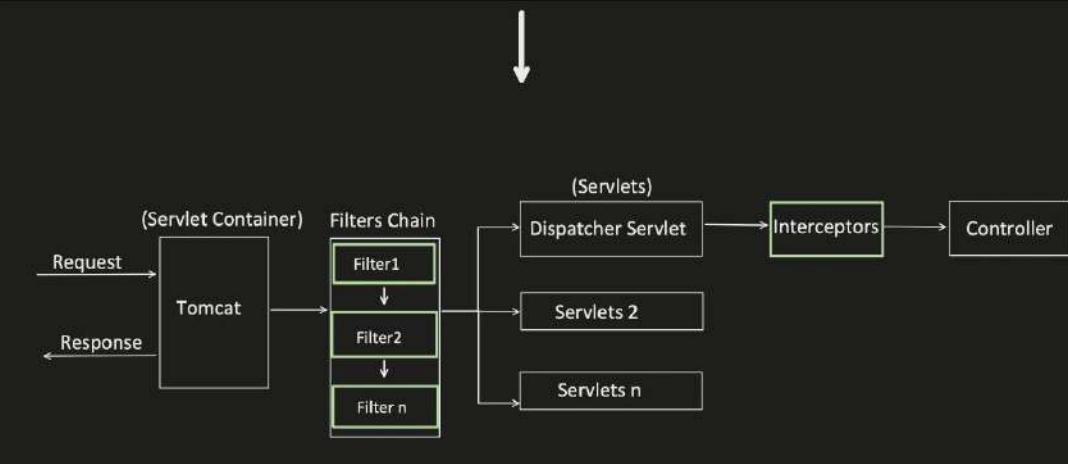
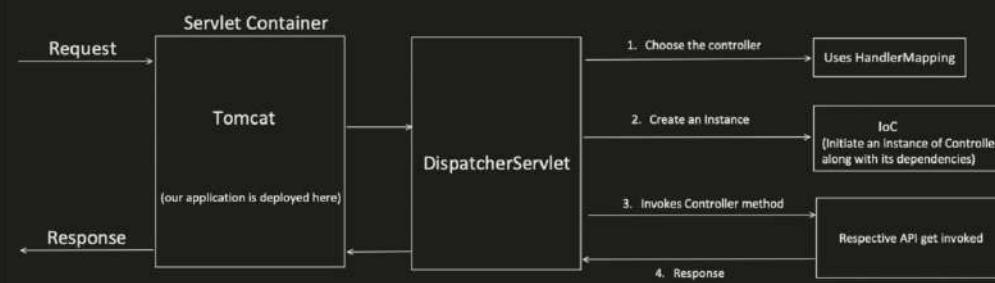
### Filter:

It intercepts the HTTP Request and Response, before they reach to the servlet.

### Interceptor:

It's specific to Spring framework, and intercepts HTTP Request and Response, before they reach to the Controller.

Lets see this diagram again:



**What is servlet:**

Servlet is nothing but a Java class, which accepts the incoming request, process it and returns the response.

We can create multiple servlets like:

Servlet 1: can be configured to handle REST APIs

Servlet 2 : can be configured to handle SOAP APIs etc....

Similarly like this, "**DispatcherServlet**" is kind of servlet provided by spring, and by default its configured to handle all APIs "/\*".

**Filter:**

Is used when we want to intercept HTTP Request and Response and add logic agnostic of the underlying servlets.

We can have many filters and have ordering between them too.

**Interceptors:**

Is used when we want to intercept HTTP request and response and add logic specific to a particular servlet.

We can have many Interceptors and have ordering between them too.

### Multiple Interceptors and its Ordering:

In previous video, we already saw, how to add 1 interceptor.  
How "preHandle", "postHandle" and "afterCompletion" comes into the picture.

Now here, will show how to add more than 1.

Also, if "preHandle" returns false, next interceptor and controller will not get invoked itself.

```
@Configuration
public class AppConfig implements WebMvcConfigurer {

    @Autowired
    MyCustomInterceptor1 myCustomInterceptor1;

    @Autowired
    MyCustomInterceptor2 myCustomInterceptor2;

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        //below order is maintained while calling interceptors
        registry.addInterceptor(myCustomInterceptor1)
            .addPathPatterns("/api/*") // Apply to these URL patterns
            .excludePathPatterns("/api/updateUser", "/api/deleteUser"); // Exclude for these URL patterns

        registry.addInterceptor(myCustomInterceptor2)
            .addPathPatterns("/api/*") // Apply to these URL patterns
            .excludePathPatterns("/api/updateUser");
    }
}
```

### Output:

---

```
2024-09-02T17:01:11.719+05:30  INFO 7162 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet
2024-09-02T17:01:11.720+05:30  INFO 7162 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet
inside pre handle Method - MyCustomInterceptor1
inside pre handle Method - MyCustomInterceptor2
hitting db to get the userdata
inside post handle method- MyCustomInterceptor2
inside post handle method- MyCustomInterceptor1
inside after completion method - MyCustomInterceptor2
inside after completion method- MyCustomInterceptor1
|
```

```

How to Add Filters:

import jakarta.servlet.*;
import java.io.IOException;

public class MyFilter1 implements Filter {
    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
        Filter.super.init(filterConfig);
    }

    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain)
            throws IOException, ServletException {
        System.out.println("MyFilter1 inside");
        filterChain.doFilter(servletRequest, servletResponse);
        System.out.println("MyFilter1 completed");
    }

    @Override
    public void destroy() {
        Filter.super.destroy();
    }
}

@Configuration
public class AppConfig {
    @Bean
    public FilterRegistrationBean<MyFilter1> myFilter1Registration() {
        FilterRegistrationBean<MyFilter1> filterRegistrationBean = new FilterRegistrationBean<MyFilter1>();
        filterRegistrationBean.setFilter(new MyFilter1());
        filterRegistrationBean.addPattern("/*");
        filterRegistrationBean.setName("MyFilter1");
        return filterRegistrationBean;
    }

    @Bean
    public FilterRegistrationBean<MyFilter2> myFilter2Registration() {
        FilterRegistrationBean<MyFilter2> filterRegistrationBean = new FilterRegistrationBean<MyFilter2>();
        filterRegistrationBean.setFilter(new MyFilter2());
        filterRegistrationBean.addPattern("/*");
        filterRegistrationBean.setName("MyFilter2");
        return filterRegistrationBean;
    }
}

```

**Output:**

-----

```

2024-09-02T17:30:39.279+05:30  INFO 7350 --- [nio-8080-exec-1]
2024-09-02T17:30:39.280+05:30  INFO 7350 --- [nio-8080-exec-1]
MyFilter2 inside
MyFilter1 inside
hitting db to get the userdata
MyFilter1 completed
MyFilter2 completed

```

If both Interceptor and Filter used together, Output will look like this.

```

2024-09-02T17:32:25.643+05:30  INFO 7373 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet
2024-09-02T17:32:25.644+05:30  INFO 7373 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet
MyFilter1 inside
MyFilter2 inside
inside pre handle Method - MyCustomInterceptor1
inside pre handle Method - MyCustomInterceptor2
hitting db to get the userdata
inside post handle method- MyCustomInterceptor2
inside post handle method- MyCustomInterceptor1
inside after completion method - MyCustomInterceptor2
inside after completion method- MyCustomInterceptor1
MyFilter2 completed
MyFilter1 completed

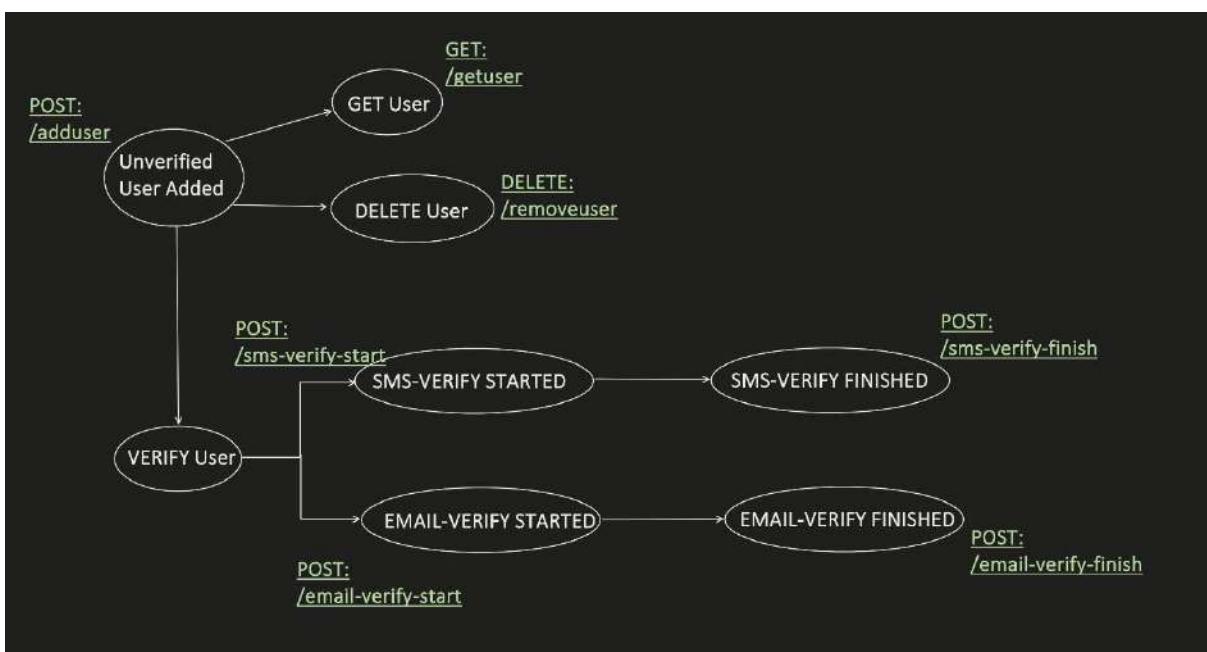
```

## Spring boot: HATEOAS

**HATEOAS**  
**Hypermedia As The Engine Of Application State**

It tells the client, what the next action you can perform on particular item.

For example:



API Response, after "*"Unverified User Added"* POST: /adduser API

Without HATEOAS link

```
{ "userID": "123456", "name": "SJ", "verifyStatus": "UNVERIFIED" }
```

With HATEOAS link

```
{ "userID": "123456", "name": "SJ", "verifyStatus": "UNVERIFIED", "links": [ { "rel": "self", "href": "http://localhost:8080/api/getUser/123456", "type": "GET" } ] }
```

Before understanding **HOW** to do this, lets understand **WHEN** to use and **WHY** to use HATEOS link?

2 Major Purpose of using HATEOAS link is to achieve

- "LOOSE COUPLING" and
- "API DISCOVERY"

To achieve above, server provides the next set of APIs (actions) in the Response itself, which client can take. So that client have less business logic around APIs (which API to invoke, when to invoke, how to invoke etc....)

But, Adding all next set of ACTIONS can make our API Response Bloat up and has several disadvantages:

- Increase complexity at server side.
- Latency impact.
- Increase Payload size.

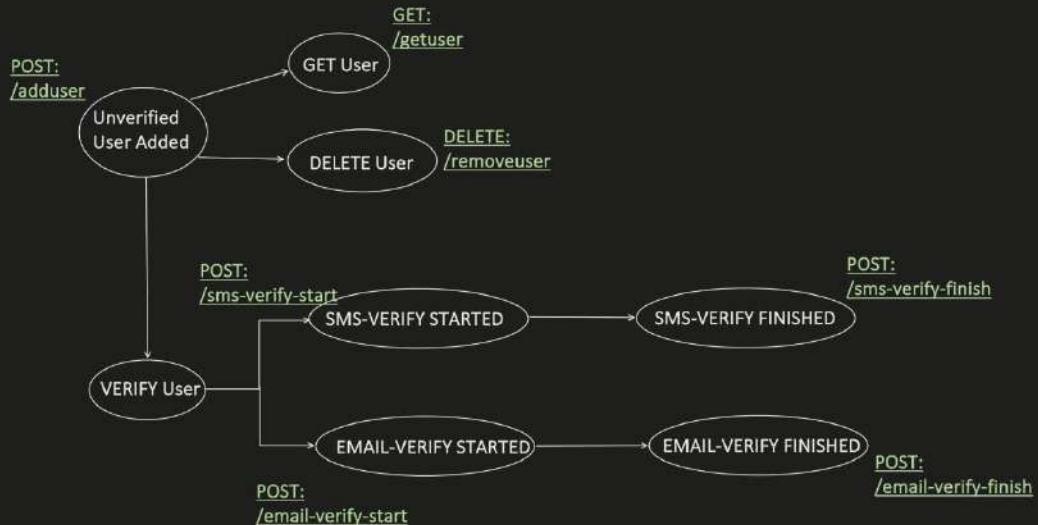
```
{  
    "userID": "123456",  
    "name": "S2",  
    "verifyStatus": "UNVERIFIED",  
    "links": [  
        {  
            "rel": "self",  
            "href": "http://localhost:8080/api/getUser/123456",  
            "type": "GET"  
        },  
        {  
            "rel": "remove",  
            "href": "http://localhost:8080/api/removeuser/123456",  
            "type": "DELETE"  
        },  
        {  
            "rel": "update",  
            "href": "http://localhost:8080/api/updateuser/123456",  
            "type": "PATCH"  
        },  
        {  
            "rel": "verify-start",  
            "href": "http://localhost:8080/api/sms-verify-start/123456",  
            "type": "POST"  
        },  
        {  
            "rel": "verify-finish",  
            "href": "http://localhost:8080/api/sms-verify-finish/123456",  
            "type": "POST"  
        }  
    ]  
}
```

Never Ever add all possible next set of actions (API) just like that.



So, proper analysis need to be done, what actually will help us to achieve **LOOSE COUPLING**

Now, if we see the above diagram again



I see, the tight coupling lies during VERIFY process.

Client need some info, before it can decide which Verify API to invoke. For example:

```
{  
    "userID": "123456",  
    "name": "SJ",  
    "verifyStatus": "UNVERIFIED",  
    "verifyType": "SMS",  
    "verifyState": "NOT_YET_STARTED"  
}
```

Client need to put business logic, that

```
if(VerifyStatus == "UNVERIFIED")  
{  
    if(verifyType == "SMS")  
    {  
        if(verifyState == "NOTE_YET_STARTED")  
        {  
            Call POST: /sms-verify-start  
        }  
        Else if (verifyState == "STARTED")  
        {  
            Call POST: /sms-verify-finish  
        }  
    }  
    Else if(verifyType == "EMAIL")  
    {  
        if(verifyState == "NOTE_YET_STARTED")  
        {  
            Call POST: /email-verify-start  
        }  
        Else if (verifyState == "STARTED")  
        {  
            Call POST: /email-verify-finish  
        }  
    }  
}
```

This dependency, can be removed by HATEOAS link

```
{  
    "userID": "123456",  
    "name": "SJ",  
    "verifyStatus": "UNVERIFIED",  
    "links": [  
        {  
            "rel": "verify",  
            "href": "http://localhost:8080/api/sms-verify-finish/123456",  
            "type": "POST"  
        }  
    ]  
}
```

Now, we have achieved LOOSE COUPLING and  
Client code looks like this.

```
if(verifyStatus == "UNVERIFIED") {  
    //invoke the verify URL, given in  
    //HATEOAS link  
}
```

## Dependency Required:

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-hateoas</artifactId>
<version>2.6.4</version>
</dependency>
```

```
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @Autowired
    User user;

    @PostMapping(path = "/adduser")
    public ResponseEntity<UserResponse> adduser() {
        UserResponse response = user.getUser();

        //our business logic to determine which Verify API need to be invoked.
        Link verifyLink = WebMvcLinkBuilder.linkTo(UserController.class).WebMvcLinkBuilder
            .slash("msc-verify-finish")
            .slash(response.getUserId())
            .withRel("verify").link()
            .withType("POST");

        response.addLink(verifyLink);

        return new ResponseEntity<>(response, HttpStatus.OK);
    }
}

public class HateoasLinks {
    private List<Link> links = new ArrayList<>();

    public void addLink(Link link) {
        links.add(link);
    }
}

public class UserResponse extends HateoasLinks {
    private String userId;
    private String name;
    private String verifyStatus;

    //getters and setters here
}
```

Other way to create Link:

```
Link verifyLink = Link.of(href: "/api/sms-verify-finish/" + response.getUserID())
    .withRel(relation: "verify")
    .withType("POST");
```

```
{
  "userID": "123456",
  "name": "SJ",
  "verifyStatus": "UNVERIFIED",
  "links": [
    {
      "rel": "verify",
      "href": "http://localhost:8080/api/sms-verify-finish/123456",
      "type": "POST"
    }
  ]
}
```

## Spring boot: ResponseEntity and Codes

Response Generally contains 3 parts:

Status Code: HTTP return code like 200 OK, 500 INTERNAL\_ERROR etc.  
Header: Additional information (Optional)  
Body: Data need to be sent in response.

We can use "ResponseEntity<T>" to create Response and in this 'T' represents the type of the 'Body'.

```
@GetMapping (path = "/get-user")
public ResponseEntity<String> getUser() {

    return ResponseEntity.ok("My Response body Object can go here");
}
```

OR

```
@GetMapping (path = "/get-user")
public ResponseEntity<String> getUser() {

    HttpHeaders headers = new HttpHeaders();
    headers.add(headerName: "My-Header1", headerValue: "SomeValue1");
    headers.add(headerName: "My-Header2", headerValue: "SomeValue2");

    return ResponseEntity.status(HttpStatus.OK)
        .headers(headers)
        .body("My Response body Object can go here");
}
```

body should be last, actually its kind of using Builder design pattern, so 'status' , 'headers' all are returning Builder object and 'body' method call returns the ResponseEntity object.

```
public static BodyBuilder status(HttpStatusCode status) {  
    Assert.notNull(status, message:"HttpStatusCode must not be null");  
    return new DefaultBuilder(status);  
}  
  
B headers(@Nullable HttpHeaders headers);  
  
<T> ResponseEntity<T> body(@Nullable T body);
```

So, what to do, when we don't want to add any body in the response:

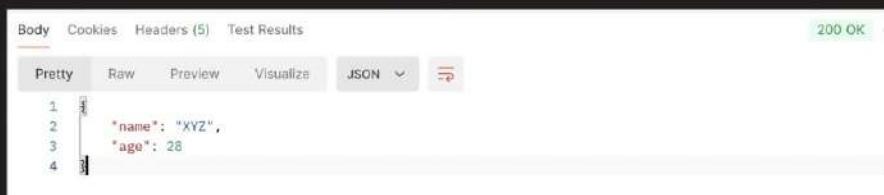
Then, we should use 'build' method

```
<T> ResponseEntity<T> build();  
  
@GetMapping (path = "/get-user")  
public ResponseEntity<Void> getUser() {  
  
    HttpHeaders headers = new HttpHeaders();  
    headers.add( headerName: "My-Header1" , headerValue: "SomeValue1" );  
    headers.add( headerName: "My-Header2" , headerValue: "SomeValue2" );  
  
    return ResponseEntity.status(HttpStatusCode.OK)  
        .headers(headers).build();  
}
```

by-default, 200 Ok is the status code set:

```
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @GetMapping (path = "/get-user")
    public User getUser() {
        User responseObj = new User( name: "XYZ", age: 28);
        return responseObj;
    }
}
```



### **@ResponseBody**

When we return Plain string or POJO directly from the class, then **@ResponseBody** annotation is required.

Why?

It tells to consider value as Response Body and not the View.

But in above example (mentioning below also), we did not use **@ResponseBody**

```
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @GetMapping (path = "/get-user")
    public User getUser() {
        User responseObj = new User( name: "XYZ", age: 28);
        return responseObj;
    }
}
```

Its because, `@RestController`, automatically puts `@ResponseBody` to all the methods

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Controller
@ResponseBody
public @interface RestController {
    @AliasFor(
        annotation = Controller.class
    )
    String value() default "";
}
```

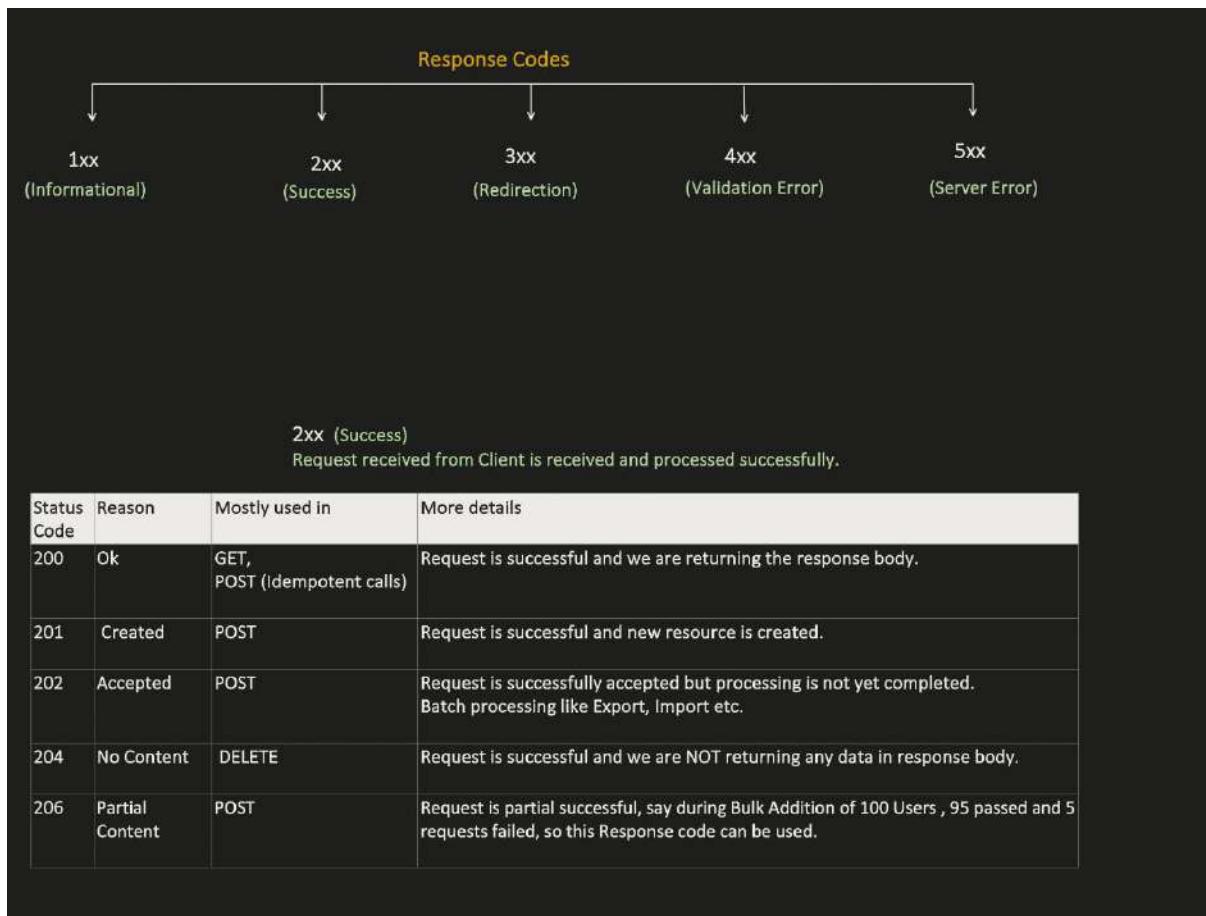
But lets say, if you use `@Controller` instead, then below code will throw exception

```
@Controller
@RequestMapping(value = "/api/")
public class UserController {

    @GetMapping(path = "/get-user")
    public String getUser() {
        return "XYZ";
    }
}
```



Because, Return value is treated as "view" and spring boot will try to look for file with the given name "XYZ" which do not exist.



**3xx (Redirection)**  
Client must take additional action to complete the request

Status Code	Reason	Mostly used in	More details
301	Moved Permanently	When we migrate from Legacy API to new API (Old Status code, new one is 308)	All request should directed to the new URI.
308	Permanent Redirect	When we migrate from Legacy API to new API	Same as 301, but it do not allow HTTP Method to change while redirect (for ex: if Old API call is POST, then NEW API should also be POST, which is relaxed in 301)
304	Not Modified	GET  PATCH <span style="color: red;">X</span>  (try to avoid using it with PATCH for example: you trying to update a name of the USER, but lets say name is already same in the DB, so no update required. In that case, we should not throw 304 (NOT_MODIFIED) error code, instead 204(NO_CONTENT) or 200(OK) is more appropriate.)	<ol style="list-style-type: none"> <li>Client makes a GET call, Server returned it with Last-Modified time in header.</li> <li>Client cache the response.</li> <li>Client make a GET call, pass this Last modified time in "If-Modified-Since" header.</li> <li>Server check the particular resource last update time with what client provided, if resource is not updated, server simply returns 304 (NOT_MODIFIED).</li> <li>If Modified, server process the request as usual and returns the new values.</li> </ol>

**4xx (Validation Errors)**  
Client need to pass correct request to server

Status Code	Reason	Mostly used in	More details
400	Bad Request	GET, POST, PATCH, DELETE	Client is not passing the required details to process the request.
401	Unauthorized	GET, POST, PATCH, DELETE	Any API, which require Authentication(like Bearer token, Basic authentication etc..) and client try to access it without providing authentication details.
403	Forbidden	GET, POST, PATCH, DELETE	Lets say, only ADMIN can perform certain operation. But if API get invoked apart from ADMIN. We should throw 401 status code as clients (apart from ADMIN) do not have permission to access the resource.
404	Not Found	GET, PATCH, DELETE	The requested resource which client passes, is not found in DB by the server. For ex: GET the user details with ID: 123, but in DB there is not such ID present.
405	Method Not Allowed	GET, POST, PATCH, DELETE	Ex: Hitting GET API, but with POST HTTP Method.  In Springboot, dispatcher servlet might throw this error, as control not even reach to controller.
422	Un-processable Entity	GET, POST, PATCH, DELETE	Your application Business validation: Like France Users should not be allowed to open an account. (as country is not supported yet)
429	Too Many Requests	GET, POST, PATCH, DELETE	Lets say: our rule is: 1 user can max make 10 calls in a minute. if User:12345 makes the 11th call in a minute then this 11th call should get failed and we can throw 429 error code.

**5xx (Server Errors)**  
Request got failed at Server, even though client passed the valid request. Means Something wrong at Server.

Status Code	Reason	Mostly used in	More details
500	Internal Server Error	GET, POST, PATCH, DELETE	Generic error code when no more specific error code is suitable.
501	Not Implemented	GET, POST, PATCH, DELETE	API lacks the ability to fulfill the request. Or say, API is in development and in future it will be available.
502	Bad Gateway	GET, POST, PATCH, DELETE	Server acting as a proxy and while calling upstream got invalid response.  Example: My application is deployed behind Reverse Proxy (Nginx). If NginX is not able to communicate with my application (because of misconfiguration of port number or something), then it is eligible to throw 502 Bad Gateway.

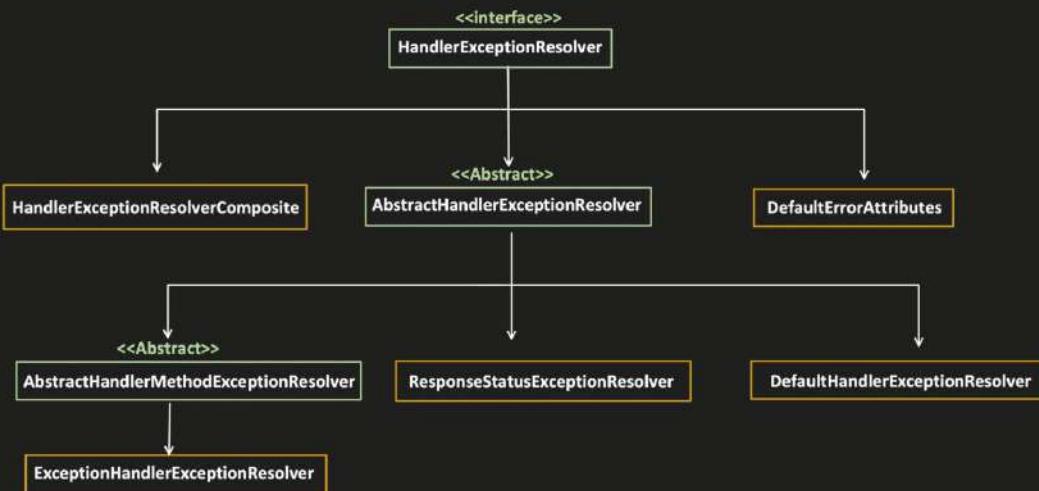
**1xx (Informational)**

Interim response to communicate request progress or its status before processing the final request.

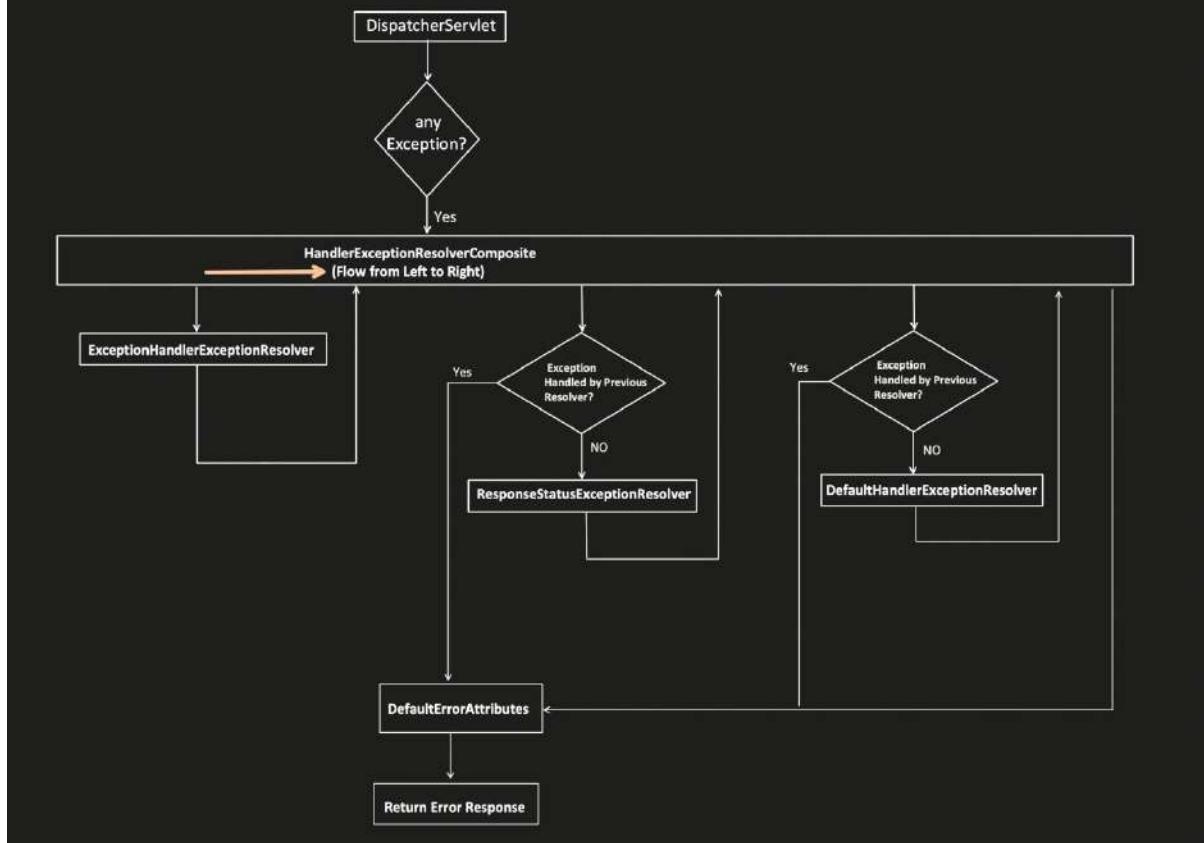
Status Code	Reason	Mostly used in	More details
100	Continue	POST	<p>Before sending the request, client check with server, if it can handle the request and ready:</p> <p>1. Client add few things in the header first, like:</p> <ul style="list-style-type: none"><li>- content length : 1048576</li><li>- content type : multipart/form-data</li><li>- Expect: 100-continue</li></ul> <p>2. Server, checks that in header, 'Expect:100-continue' is present, means, client is just checking. So server validate everything (authentication, authorization, content type, length etc. )</p> <p>3. If Server is okay, it return 100 CONTINUE status code</p> <p>4. Client receives it and then invokes the API again without Expect and server process the request.</p>

## Springboot : Exception Handling

Classes involved in handling an Exception:



Let's understand the sequence, when any exception occurs:



Let's see the flow with an example:

```
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @GetMapping(path = "/get-user")
    public String getUser() {
        throw new NullPointerException("throwing null pointer exception for testing");
    }
}
```

Output:

The screenshot shows a REST API testing interface. The URL is set to `localhost:8080/api/get-user`. In the 'Params' tab, there are two query parameters: 'Expect' with value '100' and 'Key'. The 'Body' tab displays a JSON response with the following content:

```
1  {
2      "timestamp": "2024-10-22T16:36:34.796+00:00",
3      "status": 500,
4      "error": "Internal Server Error",
5      "path": "/api/get-user"
6  }
```

A red box highlights the status code `500 Internal Server Error`.

```

@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @GetMapping(path = "/get-user")
    public String getUser() {
        throw new CustomException(HttpStatus.BAD_REQUEST,
                "request is not correct, UserID is missing");
    }
}

```

```

public class CustomException extends RuntimeException {

    HttpStatus status;
    String message;

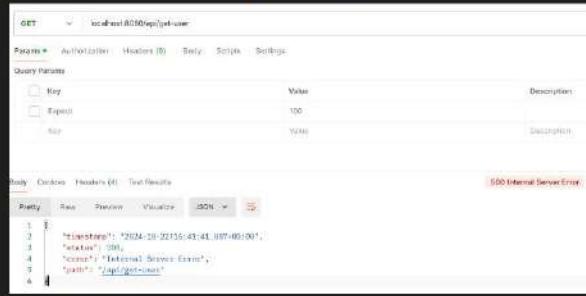
    CustomException(HttpStatus status, String message) {
        this.status = status;
        this.message = message;
    }

    public HttpStatus getStatus() {
        return status;
    }

    @Override
    public String getMessage() {
        return message;
    }
}

```

again output is same:



Why for both output is same?

Why my Return Code "BAD\_REQUEST" i.e. 400 and my error message is not shown in output?

In both the example, we are not creating the **ResponseType** object, we are just returning the exception, some other class is creating the **ResponseType** Object.

If we need full control and don't want to rely on Exception Resolvers, then we have to create the **ResponseType** Object.

```

@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @GetMapping(path = "/get-user")
    public ResponseEntity<ErrorResponse> getUser() {
        try {
            //your business logic here
            throw new CustomException(HttpStatus.BAD_REQUEST, "UserID is missing");
        } catch (CustomException e) {
            ErrorResponse errorResponse = new ErrorResponse(e.getDate(), e.getMessage(), e.getStatus().value());
            return new ResponseEntity<ErrorResponse>(errorResponse, HttpStatus.INTERNAL_SERVER_ERROR);
        }
    }
}

public class CustomException extends RuntimeException {
    private Date timestamp;
    private String msg;
    private int status;

    public ErrorResponse getDate() {
        this.msg = msg;
        this.status = status;
        this.timestamp = timestamp;
    }

    public Date getTimestamp() {
        return timestamp;
    }

    public String getMessage() {
        return msg;
    }

    public int getStatus() {
        return status;
    }

    public void setTimestamp(Date timestamp) {
        this.timestamp = timestamp;
    }

    public void setStatus(int status) {
        this.status = status;
    }
}

```

Body: {"timestamp": "2024-10-20T12:13:42.918+00:00", "status": 400, "message": "UserID is missing"}

400 Bad Request

So, When we don't return ResponseEntity like below:

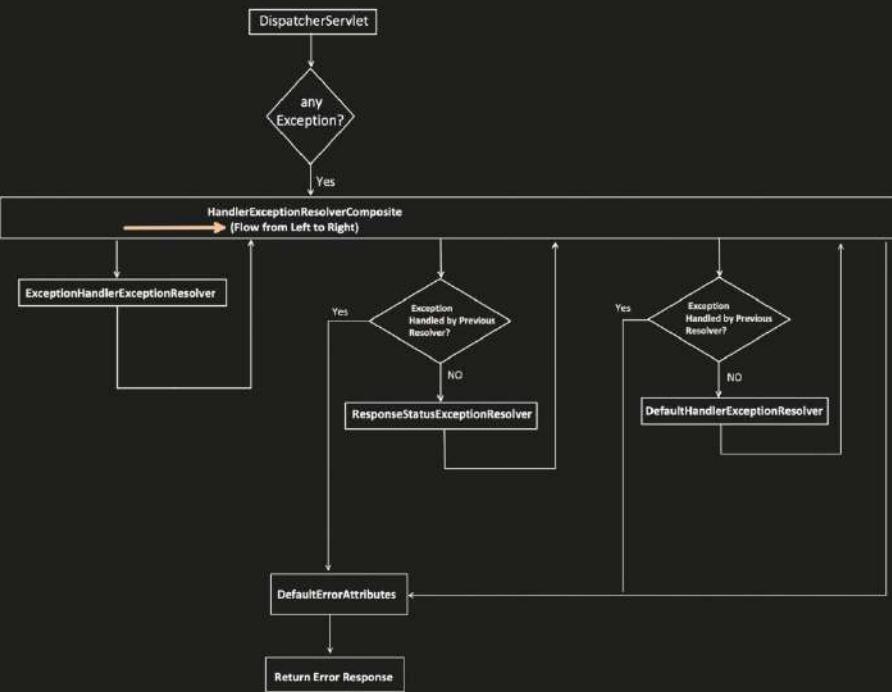
```

@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @GetMapping(path = "/get-user")
    public String getUser() {
        throw new CustomException(HttpStatus.BAD_REQUEST,
                               "request is not correct, UserID is missing");
    }
}

```

then in Exception scenario, exception passes through each Resolver like "ExceptionHandlerExceptionResolver", "ResponseStatusExceptionResolver" and "DefaultHandlerExceptionResolver" in sequence.



Each Resolver set the proper **status** and **message** in HTTP response for exceptions which they are taking care of.  
and

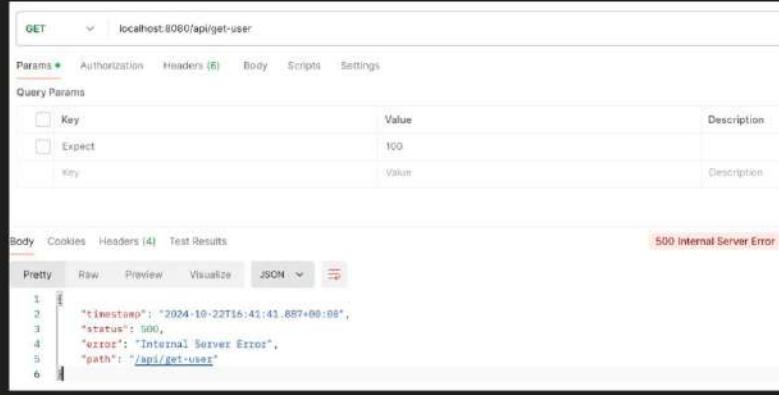
`NullPointerException` and `CustomException` all the 3 Resolvers, do not understand, so Status and Message is not set.

So, when control reaches to "**DefaultErrorAttributes**" class, it fills the values in **HTTP Response** with **default values**.

## DefaultErrorAttributes

```
@Override  
public Map<String, Object> getErrorAttributes(WebRequest webRequest, ErrorAttributeOptions options) {  
    Map<String, Object> errorAttributes = getErrorAttributes(webRequest, options.isIncluded(Include.STACK_TRACE));  
    options.retainIncluded(errorAttributes);  
    return errorAttributes;  
}  
  
private Map<String, Object> getErrorAttributes(WebRequest webRequest, boolean includeStackTrace) {  
    Map<String, Object> errorAttributes = new LinkedHashMap<>();  
    errorAttributes.put("timestamp", new Date());  
    addStatus(errorAttributes, webRequest);  
    addErrorDetails(errorAttributes, webRequest, includeStackTrace);  
    addPath(errorAttributes, webRequest);  
    return errorAttributes;  
}
```

```
@RequestMapping  
public ResponseEntity<Map<String, Object>> error(HttpServletRequest request) {  
    HttpStatus status = this.getStatus(request);  
    if (status == HttpStatus.NO_CONTENT) {  
        return new ResponseEntity(status);  
    } else {  
        Map<String, Object> body = this.getErrorAttributes(request, this.getErrorAttributeOptions(request, MediaType.ALL));  
        return new ResponseEntity(body, status);  
    }  
}
```



The screenshot shows a POSTMAN API request configuration for a GET request to `localhost:8080/api/get-user`. The 'Params' tab is selected, containing a single parameter `Expect` with value `100`. The 'Body' tab is selected, showing a JSON response with the following content:

```
1 | {  
2 |     "timestamp": "2024-10-22T16:41:41.887+00:00",  
3 |     "status": 500,  
4 |     "error": "Internal Server Error",  
5 |     "path": "/api/get-user"  
6 | }
```

The status bar at the bottom right indicates a `500 Internal Server Error`.

So, now question is: what exception does "ExceptionHandlerExceptionResolver", "ResponseStatusExceptionResolver" and "DefaultHandlerExceptionResolver" handles?

### 1. ExceptionHandlerExceptionResolver

Responsible for handling below annotations:

- `@ExceptionHandler`
- `@ControllerAdvice`

Controller level Exception handling:

```
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @GetMapping(path = "/get-user")
    public ResponseEntity<?> getUser() {
        //your business logic and validations...
        throw new CustomException(HttpStatus.BAD_REQUEST, "UserID is missing");
    }

    @ExceptionHandler(CustomException.class)
    public ResponseEntity<String> handleCustomException(CustomException ex) {
        return new ResponseEntity<>(ex.getMessage(), ex.getStatus());
    }
}
```



Use-case just to show returning Error Response object instead of just message:

```

@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @GetMapping(path = "/get-user")
    public ResponseEntity<User> getUser() {
        //your business logic and validations...
        throw new CustomException(HttpStatus.BAD_REQUEST, "UserID is missing");
    }

    @ExceptionHandler(CustomException.class)
    public ResponseEntity<Object> handleCustomException(CustomException ex) {
        ErrorResponse errorResponse = new ErrorResponse(new Date(), ex.getMessage(), ex.getStatus().value());
        return new ResponseEntity<Object>(errorResponse, ex.getStatus());
    }
}

```

```

public class ErrorResponse {

    private Date timestamp;
    private String msg;
    private int status;

    public ErrorResponse(Date timestamp, String msg, int status) {
        this.timestamp = timestamp;
        this.msg = msg;
        this.status = status;
        this.timestamp = timestamp;
    }

    public Date getTimestamp() {
        return timestamp;
    }

    public String getMessage() {
        return msg;
    }

    public int getStatus() {
        return status;
    }
}

```

The screenshot shows a REST API tool interface. At the top, there are tabs for Body, Cookies, Headers (4), and Test Results. The Body tab is selected and displays a JSON response:

```

1: {
2:   "timestamp": "2024-10-24T15:41:24.294+00:00",
3:   "status": 400,
4:   "message": "UserID is missing"
5: }

```

At the top right, it says "400 Bad Request".

Use-case just to show multiple @ExceptionHandler in single Controller class:

```

@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @GetMapping(path = "/get-user")
    public ResponseEntity<User> getUser() {
        //your business logic and validations...
        throw new CustomException(HttpStatus.BAD_REQUEST, "UserID is missing");
    }

    @GetMapping(path = "/get-user-history")
    public ResponseEntity<Object> getUserHistory() {
        //your business logic and validations...
        throw new IllegalArgumentException("Inappropriate arguments passed");
    }

    @ExceptionHandler(CustomException.class)
    public ResponseEntity<String> handleCustomException(CustomException ex) {
        return new ResponseEntity<String>(ex.getMessage(), ex.getStatus());
    }

    @ExceptionHandler(IllegalArgumentException.class)
    public ResponseEntity<String> handleCustomException(IllegalArgumentException ex) {
        return new ResponseEntity<String>(ex.getMessage(), HttpStatus.BAD_REQUEST);
    }
}

```

The screenshot shows two separate REST API requests in a tool like Postman.

**Request 1:** URL: localhost:8080/api/get-user-history

Method: GET

Headers: Content-Type: application/json

Body: (empty)

Result: 400 Bad Request

Body: { "message": "Inappropriate arguments passed" }

**Request 2:** URL: localhost:8080/api/get-user

Method: GET

Headers: Content-Type: application/json

Body: (empty)

Result: 400 Bad Request

Body: { "message": "UserID is missing" }

Use-case just to show 1 @ExceptionHandler handling multiple exceptions:

```
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @GetMapping(path = "/get-user")
    public ResponseEntity<User> getUser() {
        //your business logic and validations...
        throw new CustomException(HttpStatus.BAD_REQUEST, "UserID is missing");
    }

    @GetMapping(path = "/get-user-history")
    public ResponseEntity<UserHistory> getUserHistory() {
        //your business logic and validations...
        throw new IllegalArgumentException("inappropriate arguments passed");
    }

    @ExceptionHandler({CustomException.class, IllegalArgumentException.class})
    public ResponseEntity<String> handleCustomException(Exception ex) {
        return new ResponseEntity<String>(ex.getMessage(), HttpStatus.BAD_REQUEST);
    }
}
```

The image contains two side-by-side screenshots of a REST client interface. Both screenshots show a 'GET' request to a specific endpoint. The left screenshot's endpoint is '/localhost:8080/get-user' and the right one is '/localhost:8080/get-user-history'. Both requests result in a '400 Bad Request' status code. The left response body is '1. UserID is missing' and the right one is '1. inappropriate arguments passed'. This demonstrates that a single @ExceptionHandler can handle multiple exception types and return different messages based on the exception type.

Use-case just to show @ExceptionHandler not returning ResponseEntity and let "DefaultErrorAttributes" to create the ResponseEntity

```
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @GetMapping(path = "/get-user")
    public ResponseEntity<User> getUser() {
        //your business logic and validations...
        throw new CustomException(HttpStatus.BAD_REQUEST, "UserID is missing");
    }

    @GetMapping(path = "/get-user-history")
    public ResponseEntity<UserHistory> getUserHistory() {
        //your business logic and validations...
        throw new IllegalArgumentException("inappropriate arguments passed");
    }

    @ExceptionHandler(CustomException.class)
    public void handleCustomException(HttpStatus response, CustomException ex) throws IOException {
        response.sendError(response.value(), ex.getMessage());
    }
}
```

application.properties

Without this DefaultErrorAttributes, filter out the message field in response



### Global Exception handling:

Problem with Controller level @ExceptionHandler is:

- if multiple controller has the same type of Exceptions then same handling we might do in multiple controller
- which is nothing but a code duplication.

```
@ControllerAdvice
public class GlobalExceptionHandling {

    @ExceptionHandler(CustomException.class)
    public ResponseEntity<String> handleCustomException(CustomException ex) {
        return new ResponseEntity<>(ex.message, ex.getStatus());
    }
}
```



What if, I provide both Controller level and Global level @ExceptionHandler, which one has more priority?

```
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @GetMapping(path = "/get-user")
    public ResponseEntity<> getUserId() {
        //your business logic and validations...
        throw new CustomException(HttpStatus.BAD_REQUEST, "UserId is missing");
    }

    @ExceptionHandler(CustomException.class)
    public ResponseEntity<String> handleCustomException(CustomException ex) {
        return new ResponseEntity<>(body(ex.message + "; from Controller ExceptionHandler", ex.getStatus()));
    }
}

@ControllerAdvice
public class GlobalExceptionHandling {

    @ExceptionHandler(CustomException.class)
    public ResponseEntity<String> handleCustomException(CustomException ex) {
        return new ResponseEntity<>(body(ex.message + "; from Global ExceptionHandler", ex.getStatus()));
    }
}
```



What if there are 2 handlers which can handle an exception, which one will be given priority:

It always follow an hierarch, from bottom to up (first look for exact match if not, check for its parent and so on...)

```
@RestController
@RequestMapping(value = "/api/")
public class UserController {
    @GetMapping(path = "/get-user")
    public ResponseEntity<User> getUser() {
        //your business logic and validations...
        throw new CustomException(HttpStatus.BAD_REQUEST, "UserID is missing");
    }
}

@ControllerAdvice
public class GlobalExceptionHandling {

    @ExceptionHandler(CustomException.class)
    public ResponseEntity<String> handleCustomException(CustomException ex) {
        return new ResponseEntity<String>(ex.getMessage(), ex.getStatus());
    }

    @ExceptionHandler(RuntimeException.class)
    public ResponseEntity<String> handleRuntimeException(RuntimeException ex) {
        return new ResponseEntity<String>(ex.getMessage(), HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

## 2. ResponseStatusExceptionResolver

Handles Uncaught exception annotated with `@ResponseStatus` annotation.

Use-case1: Used above an Exception class

```
@RestController
@RequestMapping(value = "/api/")
public class UserController {
    @GetMapping(path = "/get-user")
    public ResponseEntity<User> getUser() {
        //your business logic and validations...
        throw new CustomException("UserID is missing");
    }
}

@SuppressWarnings("serial")
ResponseStatusExceptionResolver
public class CustomException extends RuntimeException {

    CustomException(String message) {
        super(message);
    }
}
```



```

@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @GetMapping(path = "/get-user")
    public ResponseEntity<?> getUser() {
        //your business logic and validations...
        throw new CustomException("UserID is missing");
    }
}

```

```

@ResponseStatus(value = HttpStatus.BAD_REQUEST, reason = "Invalid Request Passed")
public class CustomException extends RuntimeException {

    HttpStatus status;

    CustomException(HttpStatus status, String message) {
        super(message);
        this.status = status;
    }
}

```

Body Cookies Headers (4) Test Results 400 Bad Request

Pretty Raw Preview Visualize JSON

```

1. {
2.     "timestamp": "2024-10-25T13:03:50.574+00:00",
3.     "status": 400,
4.     "error": "Bad Request",
5.     "message": "Invalid Request Passed",
6.     "path": "/api/get-user"
7. }

```

#### Use-case2: Used above an @ExceptionHandler method

Again **ResponseStatusExceptionResolver** handles Uncaught exception annotated with **@ResponseStatus** annotation but if used with **@ExceptionHandler** then it will not be handled by "**ResponseStatusExceptionResolver**", it will be handled by Spring request handling mechanism itself.

```

@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @GetMapping(path = "/get-user")
    public ResponseEntity<?> getUser() {
        //your business logic and validations...
        throw new CustomException(HttpStatus.INTERNAL_SERVER_ERROR, "UserID is missing");
    }

    @ExceptionHandler(CustomException.class)
    @ResponseStatus(value = HttpStatus.BAD_REQUEST, reason = "Invalid Request Sent")
    public ResponseEntity<Object> handleCustomException(CustomException e) {
        return new ResponseEntity<Object>("{body: \"you are not authorized\", HttpStatus.FORBIDDEN}", HttpStatus.FORBIDDEN);
    }
}

```

```

public class CustomException extends RuntimeException {

    HttpStatus status;

    CustomException(HttpStatus status, String message) {
        super(message);
        this.status = status;
    }
}

```

Body Cookies Headers (4) Test Results 400 Bad Request

Pretty Raw Preview Visualize JSON

```

1. {
2.     "timestamp": "2024-10-25T14:55:53.069+00:00",
3.     "status": 400,
4.     "error": "Bad Request",
5.     "message": "Invalid Request Sent",
6.     "path": "/api/get-user"
7. }

```

```

ExceptionHandlerExceptionResolver.java
protected void doHandle(HttpServletRequest request,
                        HttpServletResponse response, HandlerExceptionResolver resolver, HandlerMethod handlerMethod, Exception exception) {
    if (exceptionHandlerMethod == null) {
        return;
    }

    if (exceptionHandlerMethod.isAnnotationPresent(OnException.class)) {
        OnException annotation = exceptionHandlerMethod.getAnnotation(OnException.class);
        exceptionHandlerMethod = annotation.value();
    }

    if (exceptionHandlerMethod != null) {
        try {
            exceptionHandlerMethod.invoke(request, response);
        } catch (Exception e) {
            logger.error("Error executing exception handler method " + exceptionHandlerMethod);
        }
    }
}

ServletInvocableHandlerMethod.java
public void invokeHandlerMethod(HttpServletRequest webRequest, HandlerMethod handlerMethod, Object... providerKeys) {
    Object returnValue = invokeForRequest(webRequest, handlerMethod, providerKeys);
    setResponseStatus(webRequest);
    if (returnValue == null) {
        if (webRequest.getAttribute(WebAttributes.ERROR_STATUS_ATTRIBUTE) != null || !newContainer.isRequestHandled()) {
            newContainer.setResponseHandler();
            return;
        }
    } else if (StringUtil.isNotEmpty(getResponseStatusName())) {
        newContainer.setResponseStatusName();
    }
    newContainer.setRequestHandled();
    Assert.state("Expected this.returnValue to be null", returnValue != null, "No return value handled");
    try {
        webRequest.setAttribute(WebAttributes.RETURN_VALUE_ATTRIBUTE, returnValue);
        newContainer.setReturnHandler();
        return;
    } catch (Exception ex) {
        if (logger.isTraceEnabled()) {
            logger.trace("Error setting return value " + returnValue);
        }
    }
}

```

What if @ExceptionHandler method, set Response status and message itself instead of returning the response entity:

```

@RestController
@RequestMapping(value = "/api/*")
public class UserController {

    @GetMapping(path = "/get-user")
    public ResponseEntity<User> getUser() {
        //your business logic and validation...
        throw new CustomException(HttpStatus.INTERNAL_SERVER_ERROR, "UserID is missing");
    }

    @ExceptionHandler(CustomException.class)
    @ResponseStatus(value = HttpStatus.BAD_REQUEST, reason = "Invalid Request Sent")
    public void handleCustomException(CustomException e, HttpServletResponse response) throws IOException {
        response.sendError(HttpStatus.FORBIDDEN.value(), e.getMessage());
    }
}

```



Its because, Response.sendError first set the status and message in response and do COMMIT.

2nd ResponseStatus method will try to do the same thing, and Exception will occur in ExceptionHandlerResolver class as we try to reset already committed status field.

So its advisable not to use together `@ExceptionHandler` and `@ResponseStatus` together to avoid confusion.

But if you have to, use like below:

```
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @GetMapping(path = "/get-user")
    public ResponseEntity<?> getUser() {
        //your business logic and validations...
        throw new CustomException(HttpStatus.INTERNAL_SERVER_ERROR, "UserID is missing");
    }

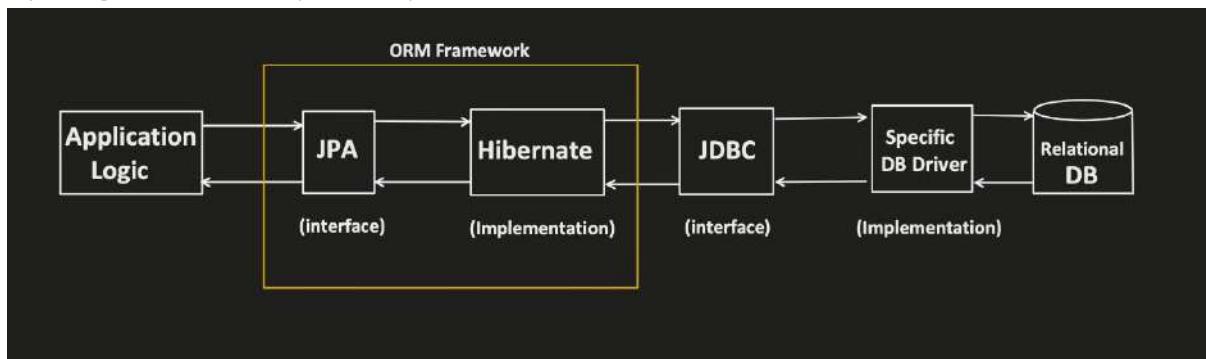
    @ExceptionHandler(CustomException.class)
    @ResponseStatus(value = HttpStatus.BAD_REQUEST, reason = "Invalid Request Sent")
    public void handleCustomException(CustomException e) {
        //do nothing here
    }
}
```



### 3. DefaultHandlerExceptionResolver

Handles Spring framework related exceptions only like `MethodNotFound`, `NoResourceFound` etc..

## Spring boot: JPA (Part-1)



### Before going to JPA, lets recall JDBC

JDBC (Java Database Connectivity) provides an **Interface** to:

- Make connection with DB
- Query DB
- and process the result

Actual implementation is provided by **Specific DB Drivers**.

For example:

MySQL

- o Driver : Connector/J
- o Class : *com.mysql.cj.jdbc.Driver*

PostgreSQL

- o Driver : PostgreSQL JDBC Driver
- o Class : *org.postgresql.Driver*

H2 (in-memory)

- o Driver : H2 Database Engine
- o Class : *org.h2.Driver*

### Using JDBC without Springboot

```
public class DatabaseConnection {
    public Connection getConnection() {
        try {
            // H2 Driver Loading
            Class.forName("org.h2.Driver");
            DB Name
            // Establish connection with DB
            return DriverManager.getConnection("jdbc:h2:mem:userDB", "user", "password");
        } catch (ClassNotFoundException | SQLException e) {
            //Handle exception
        }
        return null;
    }
}

public class UserDAO {
    public void createUserTable() {
        try {
            Connection connection = new DatabaseConnection().getConnection();
            Statement statementQuery = connection.createStatement();
            String sql = "CREATE TABLE users(user_id INT AUTO_INCREMENT PRIMARY KEY, user_name VARCHAR(100), age INT)";
            statementQuery.executeUpdate(sql);
        } catch (SQLException e) { //Handle exception here}
        finally { //close statementQuery and db connection here}
    }

    public void createuser(String userName, int userAge) {
        try {
            Connection connection = new DatabaseConnection().getConnection();
            String sqlQuery = "INSERT INTO users(user_name, age) VALUES (?, ?)";
            PreparedStatement preparedQuery = connection.prepareStatement(sqlQuery);
            preparedQuery.setString(1, userName);
            preparedQuery.setInt(2, userAge);
            preparedQuery.executeUpdate();
        } catch (SQLException e) { //Handle exception here}
        finally { //close preparedQuery and db connection here}
    }

    public void readusers() {
        try {
            Connection connection = new DatabaseConnection().getConnection();
            String sqlQuery = "SELECT * FROM users";
            PreparedStatement preparedQuery = connection.prepareStatement(sqlQuery);
            ResultSet output = preparedQuery.executeQuery();
            while (output.next()) {
                String userDetails = output.getInt("user_id") + " " +
                    output.getString("user_name") + " " +
                    output.getInt("age");
                System.out.println(userDetails);
            }
        } catch (SQLException e) { //Handle exception here}
        finally { //close preparedQuery and db connection here}
    }
}
```

If we see above example:

- Connection
- Statement
- PreparedStatement
- ResultSet etc.

All are interfaces which JDBC provide and each specific driver provide the implementation for it.

But there are so much of **BOILERCODE** present like:

- Driver class loading
- DB Connection Making
- Exception Handling
- Closing of the DB connection and other objects like Statement etc.
- Manual handling of DB Connection Pool
- Etc..

## Using JDBC with Springboot

### pom.xml

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

Springboot provides *JdbcTemplate* class, which helps to remove all the boiler code.

```
@Component
public class UserService {

    @Autowired
    UserRepository userRepository;

    public void createTable() {
        userRepository.createTable();
    }

    public void insertUser(String userName, int age) {
        userRepository.insertUser(userName, age);
    }

    public List<User> getUsers() {
        List<User> users = userRepository.getUsers();
        for(User user : users) {
            System.out.println(user.getUserId() + ":" + user.getUserName() + ":" + user.getAge());
        }
        return users;
    }
}

@Repository
public class UserRepository {

    @Autowired
    JdbcTemplate jdbcTemplate;

    public void createTable() {
        jdbcTemplate.execute("CREATE TABLE users (user_id INT AUTO_INCREMENT PRIMARY KEY, " +
                            "user_name VARCHAR(100), age INT)");
    }

    public void insertUser(String name, int age) {
        String insertQuery = "INSERT INTO users (user_name, age) VALUES (?, ?)";
        jdbcTemplate.update(insertQuery, name, age);
    }

    public List<User> getUsers() {
        String selectQuery = "SELECT * FROM users";
        return jdbcTemplate.query(selectQuery, (rs, rowNum) -> {
            User user = new User();
            user.setUserId(rs.getInt(columnLabel("user_id")));
            user.setUserName(rs.getString(columnLabel("user_name")));
            user.setAge(rs.getInt(columnLabel("age")));
            return user;
        });
    }
}
```

### **application.properties**

```
spring.datasource.url=jdbc:h2:mem:userDB
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.h2.console.enabled=true
```

```
public class User {  
  
    int userId;  
    String userName;  
    int age;  
  
    //getters and setters  
}
```

#### Driver class loading -

JdbcTemplate load it at the time of application startup in DriverManager class.

#### DB Connection Making -

jdbcTemplate takes care of it, whenever we execute any query.

#### Exception Handling -

in Plain JDBC, we get very abstracted '*SQLException*' but in jdbcTemplate, we get granular error like DuplicateKeyException, QueryTimeoutException etc.. (defined in org.springframework.dao package).

#### Closing of the DB connection and other resources -

when we invoke update or query method, after success or failure of the operation, jdbcTemplate takes care of either closing or return the connection to Pool itself.

#### - Manual handling of DB Connection Pool -

Springboot provides default jdbc connection pool i.e. '*HikariCP*' with Min and Max pool size of 10. And we can change the configuration in '*application.properties*'

```
spring.datasource.hikari.maximum-pool-size=10
spring.datasource.hikari.minimum-idle=5
```

We can also configure different jdbc connection pool if we want like below:

```

@Configuration
public class AppConfig {

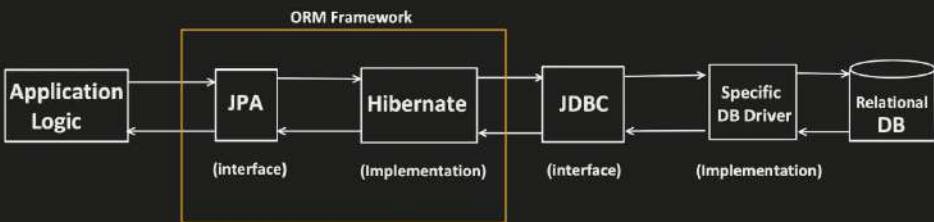
    @Bean
    public DataSource dataSource() {
        HikariDataSource dataSource = new HikariDataSource();
        dataSource.setDriverClassName("org.h2.Driver");
        dataSource.setJdbcUrl("jdbc:h2:mem:userDB");
        dataSource.setUsername("sa");
        dataSource.setPassword("");
        return dataSource;
    }
}

```

JdbcTemplate frequently used methods

Method Name	Use For	Sample
update(String sql, Object... args)	Insert Update Delete	<pre> String insertQuery = "INSERT INTO users (user_name, age) VALUES (?, ?)"; int rowsAffected = jdbcTemplate.update(insertQuery, "X", 27);  String updateQuery= "UPDATE users SET age = ? WHERE user_id = ?"; int rowsAffected = jdbcTemplate.update(updateQuery, 29, 1); </pre>
update(String sql, PreparedStatementSetter pss)	Insert Update Delete	<pre> String insertQuery= "INSERT INTO users (user_name, age) VALUES (?, ?)"; jdbcTemplate.update(insertQuery, (PreparedStatement ps)-&gt;{     ps.setString(1, "X");     ps.setInt(2, 25); });  String updateQuery= "UPDATE users SET age = ? WHERE user_id = ?"; jdbcTemplate.update(updateQuery, (PreparedStatement ps)-&gt;{     ps.setString(1, 29);     ps.setInt(2, 1); }); </pre>
query(String sql, RowMapper<T> rowMapper)	Get multiple Rows	<pre> List&lt;User&gt; users = jdbcTemplate.query("SELECT * FROM users", (rs, rowNum) -&gt; {     User user = new User();     user.setId(rs.getInt("user_id"));     user.setName(rs.getString("user_name"));     user.setAge(rs.getInt("age"));     return user; }); </pre>
queryForList(String sql, Class<T> elementType)	Get Single Column of Multiple Rows	<pre> List&lt;String&gt; userNames =     jdbcTemplate.queryForList("SELECT user_name FROM users", String.class); </pre>
queryForObject(String sql, Object[] args, Class<T> requiredType)	Get single Row	<pre> User user =     jdbcTemplate.queryForObject("SELECT * FROM users WHERE user_id = ?", new Object[]{1}, User.class); </pre>
queryForObject(String sql, Class<T> requiredType)	Get Single Value	<pre> int userCount =     jdbcTemplate.queryForObject("SELECT COUNT(*) FROM users", Integer.class); </pre>

## JPA ARCHITECTURE (PART2)



### ORM (Object-Relational Mapping)

- Act as a bridge between Java Object and Database tables.
- Unlike JDBC, where we have to work with SQL, with this, we can interact with database using Java Objects.

Lets first see, 1 happy flow first, before deep diving into JPA

```
pom.xml
```

```
application.properties
```

```
#database connection properties
spring.datasource.url=jdbc:h2:mem:userDB
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
```

```
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @Autowired
    UserDetailsService userDetailsService;

    @GetMapping(path = "/test-jpa")
    public List<UserDetails> getAllUsers() {
        UserDetails userDetails = new UserDetails(name: "xyz",
            email: "xyz@conceptcoding.com");
        userDetailsService.create(userDetails);
        return userDetailsService.getAllUsers();
    }
}
```

```
@Service
public class UserDetailsService {
    @Autowired
    private UserDetailsRepository userDetailsRepository;

    public void saveUser(UserDetails user) {
        userDetailsRepository.save(user);
    }

    public List<UserDetails> getAllUsers() {
        return userDetailsRepository.findAll();
    }
}
```

```
@Repository
public interface UserDetailsRepository extends
    JpaRepository<UserDetails, Long> {
}
```

```
@Entity
public class UserDetails {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private String email;

    // Constructors
    public UserDetails() {
    }

    public UserDetails(String name, String email) {
        this.name = name;
        this.email = email;
    }

    // Getters and setters
}
```

Output:

```
{  
  "id": 1,  
  "name": "xyx",  
  "email": "xyz@conceptandcoding.com",  
  "age": 28  
}
```

To enable the console, we can add, below two properties in "application.properties"

```
spring.h2.console.enabled=true  
spring.h2.console.path=/h2-console
```

<http://localhost:8080/h2-console>

The screenshot shows the H2 Console interface. On the left, the 'Login' dialog box is displayed with the following fields:

- Saved Settings: Generic H2 (Server)
- Setting Name: Generic H2 (Server) [Save] [Remove]
- Driver Class: org.h2.Driver
- JDBC URL: jdbc:h2:mem:userDB (highlighted with a red box)
- User Name: sa
- Password: (empty field)
- Connect [Test Connection]

A green arrow points from the JDBC URL field in the login dialog to the right-hand query results window. The right-hand window shows the H2 Database browser with the following details:

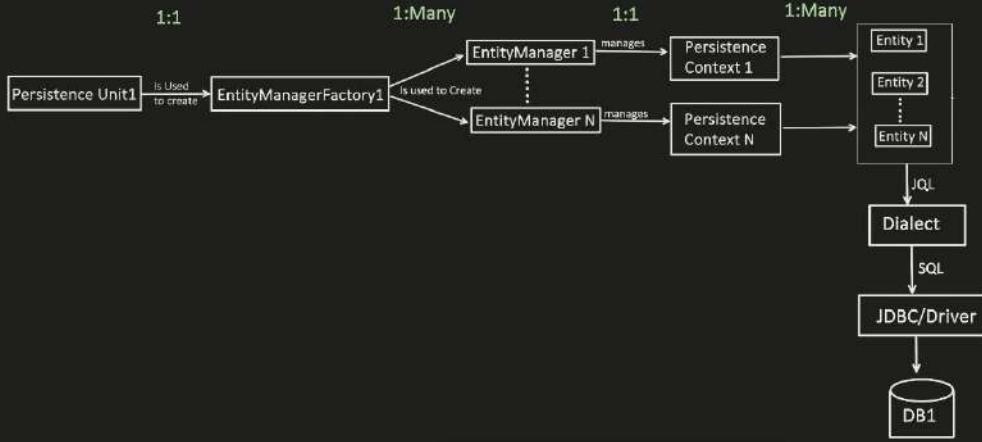
- Left sidebar: db:h2:mem:userDB, USERDETAILS, INFORMATION\_SCHEMA, Users, H2 2.2.224 (2023-06-17)
- Top menu: Run, Auto-commit, Max rows: 1900, Auto-complete: Off, DQL statement: SELECT \* FROM USERDETAILS;
- Result pane: SELECT \* FROM USERDETAILS;

ID	EMAIL	NAME
1	xyz@conceptandcoding.com	xyx

1 row, 1 ms

Looks very simple, but what's happening inside?

**JPA Architecture/Components involved:**



**pom.xml**

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

**1. Persistence Unit**

- Logical grouping of Entity classes which share same configurations.
- Configuration details like:
  - Database connection properties
  - JPA Provider (hibernate etc.) etc.

```

Persistence.xml

<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="2.0">
    <persistence-unit name="persistenceInTheHello" transaction-type="RESOURCE_LOCAL">
        <!-- Logical grouping of Entity Classes which all share same configurations -->
        <class>com.conceptandcoding.entity.SampleEntity1</class>
        <class>com.conceptandcoding.entity.SampleEntity2</class>

        <!-- Which JPA provider we want to use, Hibernate or OpenJPA or EclipseLink etc... -->
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>

        <!-- DB Connection Properties -->
        <properties>
            <!-- specific provider dialect -->
            <property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect" />

            <property name="javax.persistence.jdbc.url" value="jdbc:mysql://userDB" />
            <property name="javax.persistence.jdbc.user" value="hs" />
            <property name="javax.persistence.jdbc.password" value="123" />
            <!-- more properties here -->
        </properties>
    </persistence-unit>
</persistence>

application.properties

#database connection properties
spring.datasource.url=jdbc:h2:mem:userDB
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=

# Define packages to scan
# (its optional, spring boot looks for all @Entity, but still if we want specific package lookup
spring.jpa.packages-to-scan=com.conceptandcoding.entity

#Define provider
#(its optional, spring boot automatically picks based on provider)
spring.jpa.properties.javaee.persistence.provider=org.hibernate.jpa.HibernatePersistenceProvider
spring.jpa.database-platform=org.hibernate.dialect.MySQLDialect

#transaction type, Default is RESOURCE_LOCAL only
spring.jpa.properties.javaee.persistence.transactionType=RESOURCE_LOCAL

```

## 2. EntityManagerFactory

- Using Persistence Unit configuration, EntityManagerFactory object get created during application startup.
- If any property is not provided, default one is picked and set.
- 1 EntityManagerFactory for 1 Persistence Unit.
- This class act as a Factory to create an object of EntityManager.

### LocalContainerEntityManagerFactoryBean.java

```

@Override
protected EntityManagerFactory createNativeEntityManagerFactory() throws PersistenceException {
    Assert.state(expression: this.persistenceUnitInfo != null, message: "PersistenceUnitInfo not initialized");

    PersistenceProvider provider = getPersistenceProvider();
    if (provider == null) {
        String providerClassName = this.persistenceUnitInfo.getProviderClassName();
        if (providerClassName == null) {
            throw new IllegalArgumentException(
                "No PersistenceProvider specified in EntityManagerFactory configuration, " +
                "and chosen PersistenceUnitInfo does not specify a provider class name either");
        }
        Class<?> providerClass = ClassUtils.resolveClassName(providerClassName, getClassLoader());
        provider = (PersistenceProvider) BeanUtils.instantiateClass(providerClass);
    }

    if (logger.isDebugEnabled()) {
        logger.debug("Building JPA container EntityManagerFactory for persistence unit '" +
                    this.persistenceUnitInfo.getPersistenceUnitName() + "'");
    }
    EntityManagerFactory emf =
        provider.createContainerEntityManagerFactory(this.persistenceUnitInfo, getJpaPropertyMap());
    postProcessEntityManagerFactory(emf, this.persistenceUnitInfo);

    return emf;
}

```

What if we want to manually create an object of EntityManagerFactory?

```
@Configuration
public class AppConfig {

    @Bean
    public DataSource dataSource() {
        HikariDataSource dataSource = new HikariDataSource();
        dataSource.setDriverClassName("org.h2.Driver");
        dataSource.setJdbcUrl("jdbc:h2:mem:userDB");
        dataSource.setUsername("sa");
        dataSource.setPassword("");
        return dataSource;
    }

    @Bean
    public JpaVendorAdapter jpaVendorAdapter() {
        HibernateJpaVendorAdapter adapter = new HibernateJpaVendorAdapter();
        adapter.setGenerateDdl(true);
        adapter.setDatabasePlatform("org.hibernate.dialect.H2Dialect");
        return adapter;
    }

    @Bean
    public LocalContainerEntityManagerFactoryBean entityManagerFactory(
            DataSource dataSource,
            JpaVendorAdapter jpaVendorAdapter) {
        LocalContainerEntityManagerFactoryBean emf1 = new LocalContainerEntityManagerFactoryBean();
        emf1.setDataSource(dataSource);
        emf1.setJpaVendorAdapter(jpaVendorAdapter);
        emf1.setPackagesToScan("com.conceptandcoding.learningspringboot.jpa");
        emf1.setPersistenceUnitName("uniqueFactoryName"); // unique name for our EntityManagerFactory
        return emf1;
    }
}
```

### 3. Transaction Manager association with EntityManagerFactory

During persistence unit, we have specified the value for "transaction-type" value either:

- RESOURCE\_LOCAL (default)
- JTA (Java Transaction API)

Transaction Manager could be of 2 type:

- Manager, which managing transaction For 1 DB.
- Manager, which managing transaction which can span across multiple DB. That's also possible using JTA.

During application startup, after EntityManagerFactory object created, based on RESOURCE\_LOCAL or JTA, transaction manager object get created.

#### UseCase1: Transaction Manager for managing txn for 1 DB

application.properties

```
database.connection.properties
spring.datasource.url=jdbc:mysql://localhost:3306/test
spring.datasource.driver=com.mysql.cj.jdbc.Driver
spring.datasource.username=root
spring.datasource.password=root

# Define packages to scan
# It's optional, spring boot looks for all @Entity, but still if we want specific package (only)
spring.jpa.hibernate.packages=entity

# JPA provider
# With optional, using best automatically picks based on provider
# spring.jpa.provider=java.persistence.jdbc.JdbcPersistenceProvider
# spring.jpa.provider=hibernate.org.hibernate.HibernateJpaDialect
#spring.jpa.database=sqlserver
#spring.jpa.database=mysql

# Transaction type, default is RESOURCE_LOCAL only
#spring.jpa.hibernate.jdbc.concurrency.transaction-type=RESOURCE_LOCAL
```

JpaTransactionManager.java  
(implementation of PlatformTransactionManager)

```
@Override
public void setBeanFactory(BeanFactory beanFactory) throws BeansException {
    if (getEntityManagerFactory() == null) {
        if (!beanFactory instanceof ListableBeanFactory lbf) {
            throw new IllegalStateException("Cannot retrieve EntityManagerFactory by persistence unit name " +
                "from a non-listable BeanFactory: " + beanFactory);
        }
        setEntityManagerFactory(EntityManagerFactoryUtils.findEntityManagerFactory(lbf, getPersistenceUnitName()));
    }
}
```



The tooltip for the beanType field shows the following details:

- beanType = (Castig1008) "class org.springframework.orm.jpa.JpaTransactionManager" ... Navigate
- ↳ cachedConstructor = null
- ↳ name = "org.springframework.orm.jpa.JpaTransactionManager"
- ↳ module = (MavenModuleInfo) "unnamed module @71623276"
- ↳ classLoader = (ClientClassLoader, classloading module)
- ↳ classData = null
- ↳ packageInfo = "org.springframework.orm.jpa"
- ↳ componentType = null

We can create it manually too:

```
@Configuration
public class AppConfig {

    @Bean
    public DataSource dataSource() {
        HikariDataSource dataSource = new HikariDataSource();
        dataSource.setDriverClassName("org.h2.Driver");
        dataSource.setJdbcUrl("jdbc:h2:mem:userDB");
        dataSource.setUsername("sa");
        dataSource.setPassword("");
        return dataSource;
    }

    @Bean
    public JpaVendorAdapter jpaVendorAdapter() {
        HibernateJpaVendorAdapter adapter = new HibernateJpaVendorAdapter();
        adapter.setGenerateDdl(true);
        adapter.setDatabasePlatform("org.hibernate.dialect.H2Dialect");
        return adapter;
    }

    @Bean
    public LocalContainerEntityManagerFactoryBean entityManagerFactory(
        DataSource dataSource,
        JpaVendorAdapter jpaVendorAdapter) {
        LocalContainerEntityManagerFactoryBean emf1 = new LocalContainerEntityManagerFactoryBean();
        emf1.setDataSource(dataSource);
        emf1.setJpaVendorAdapter(jpaVendorAdapter);
        emf1.setPackagesToScan("com.conceptandcoding.learningspringboot.jpa");
        emf1.setPersistenceUnitName("uniqueFactoryName"); // unique name for our EntityManagerFactory
        return emf1;
    }

    @Bean
    public JpaTransactionManager transactionManager(EntityManagerFactory entityManagerFactory) {
        return new JpaTransactionManager(entityManagerFactory);
    }
}
```

UseCase2: Transaction Manager for creating txn which can span across multiple DB

*I will create a separate video, in which I will explain, how we can handle txn which can span across multiple database.*

*As it required some time for explaining JTA related nuisance and keywords like Atomikos, XADataSource and how it uses 2PC to orchestrate txn etc.. which is out of the context for todays topic.*

But at this point, we can understand that, TRANSACTION MANAGER is created based on what "transaction-type" we provided in the persistence unit (application.properties) file.

----- Above all steps happened during application startup, below steps happen when API gets invoked -----

#### 4. EntityManager and Persistence Context

##### **EntityManager :**

- Its an interface in JPA that provides methods to perform CRUD operations on entities.
  - ◆ **persist()** (for saving)
  - ◆ **merge()** (for updating)
  - ◆ **find()** (for fetching)
  - ◆ **remove()** (for deleting)
  - ◆ **createQuery()** (for executing JPQL queries)
- EntityManager interface methods are implemented by JPA Vendors like Hibernate etc.
- EntityManagerFactory helps to create an Object of EntityManager.

##### **PersistenceContext:**

- Consider its a first level cache.
- For each EntityManager, PersistenceContext object is created, which hold list of Entities its working on.
- Also manage the life cycle of entity.

Entity Manager Insert/Update/Delete operations are Transaction bounded. Means, it first check if 'Transaction' is open, if not, it will throw exception.

But not all (READ operations are not Transaction bounded)

```

@Service
public class UserDetailsService {

    @Autowired
    private UserDetailsRepository userDetailsRepository;

    public void saveUser(UserDetails user) {
        userDetailsRepository.save(user);
    }

    public List<UserDetails> getAllUsers() {
        return userDetailsRepository.findAll();
    }
}

```

```

@Repository
public interface UserDetailsRepository extends
    JpaRepository<UserDetails, Long> {
}

```

Internally JpaRepository all Insert, Update, Delete methods are annotated with @Transactional, so even if we do not write, spring framework takes care of it.

```

@Transactional
public <S extends T> S save(S entity) {
    Assert.notNull(entity, "Entity must not be null");
    if (this.entityInformation.isNew(entity)) {
        this.entityManager.persist(entity);
        return entity;
    } else {
        return this.entityManager.merge(entity);
    }
}

```

What if, I try to directly call EntityManager persist method, instead of spring framework.

```

@Service
public class UserDetailsService {

    @PersistenceContext
    EntityManager entityManager;

    public void saveUser(UserDetails user) {
        entityManager.persist(user);
    }
}

```

```

jakarta.persistence.TransactionRequiredException Create breakpoint : No EntityManager with actual transaction available for current thread
at org.springframework.orm.jpa.SharedEntityManagerCreator$SharedEntityManagerInvocationHandler.invoke(SharedEntityManagerCreator.java:131)
at jdk.proxy2/jdk.proxy2.$Proxy108.persist(Unknown Source) ~[na:na]
at com.conceptandcoding.learningspringboot.jpa.service.UserDetailsService.saveUser(UserDetailsService.java:23) ~[classes/:na]
at com.conceptandcoding.learningspringboot.UserController.getUser(UserController.java:22) ~[classes/:na] <4 internal lines>

```

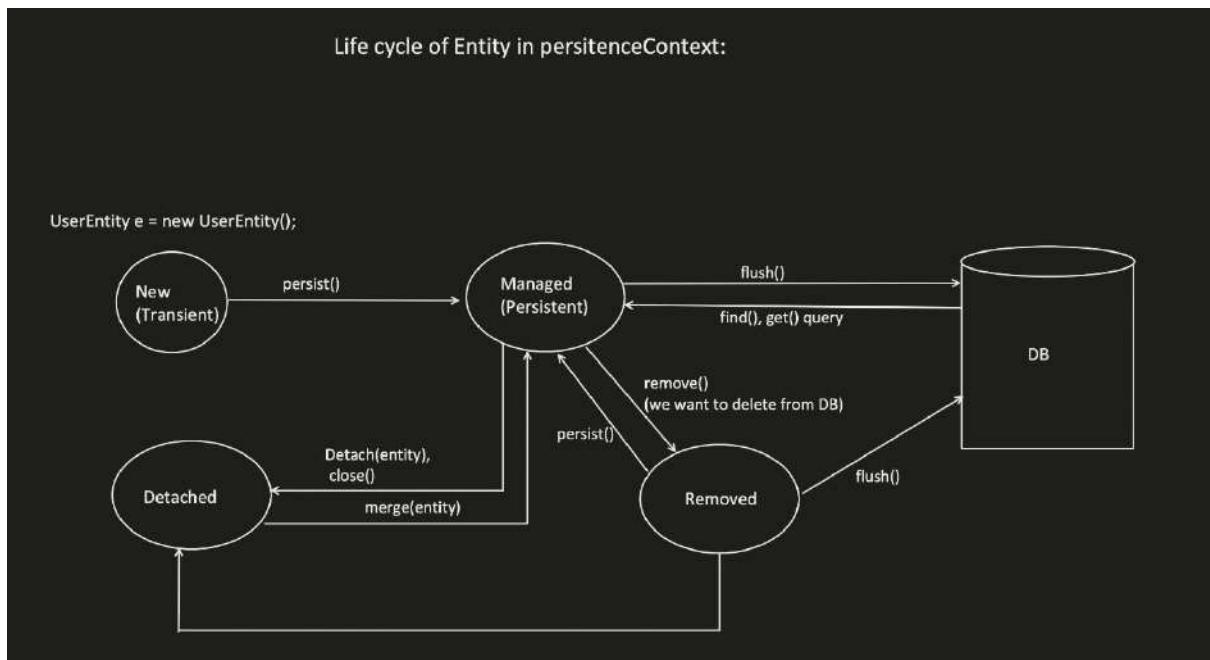
If I add `@Transactional`, it works fine now.

```
@Service
public class UserDetailsService {

    @PersistenceContext
    EntityManager entityManager;

    @Transactional
    public void saveUser(UserDetails user) {
        entityManager.persist(user);
    }
}
```

```
[{"id": 1,
  "name": "xyz",
  "email": "xyz@conceptandcoding.com"}]
```



## Springboot: First Level Caching

JPA - PART3 (First-Level Cache)

**application.properties**

```

#Database connection
spring.datasource.url=jdbc:h2:mem:user08
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=root
spring.datasource.password=

```

# Enable the H2 console and set the path
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console

spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format\_sql=true

**Controller class**

```

@RestController
@MapperMapping(path = "/api/v1")
public class UserDetailsService {

    @Autowired
    UserDetailsService implemUserDetailsService;

    @GetMapping(path = "/test-user")
    public UserDetails userDetail() {
        UserDetails userDetail = new UserDetails(name: "cvx",
                email: "vxc@com.com");
        userDetailService.createUser(userDetail);
        UserOutput output = userDetailsService.persist(userDetail);
        return output;
    }

    @GetMapping(path = "/read-user")
    public UserOutput userDetailID() {
        UserOutput output = userDetailsService.getDetail(implement.ID);
        return output;
    }
}

```

**UserDetails Entity**

```

@Entity
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private String email;

    // Constructors
    public UserDetails() {}

    public UserDetails(String name, String email) {
        this.name = name;
        this.email = email;
    }

    // Getters and setters
}

```

**UserDetailsRepository**

```

@Repository
public interface UserDetailsRepository extends JpaRepository<UserDetails, Long> {
}

```

- Internally using EntityManager API only, but it provides some predefined methods like findAll, deleteAll etc.
- Also provides pagination and sorting capability...
- Also @ Transactional support on operations which needs it on operation like Insert, modify, delete.
- Starts and closes EntityManager resource, so no need to manage it.

```

@Service
public class UserDetailsService {
    @Autowired
    UserDetailsRepository userDetailsRepository;

    public void saveUser(UserDetails user) { // Creating EntityManager
        userDetailsRepository.save(user);
    }

    public UserDetails getUser(Long primaryKey) {
        return userDetailsRepository.findById(primaryKey).get();
    }
}

SimpleJpaRepository.java
@Transactional
public <S extends T> S save(S entity) {
    Assert.notNull(entity, "Entity must not be null");
    if (this.entityInformation.isNew(entity)) {
        this.entityManager.persist(entity);
        return entity;
    } else {
        return this.entityManager.merge(entity);
    }
}

StatefulPersistenceContext.java
@Override
public void addEntity(EntityKey key, Object entity) {
    EntityHolderImpl<Holder> holder = EntityHolderImpl.<Holder>getOrCreate(key, entity);
    final EntityHolderImpl<OldHolder> oldHolder = getInitializedEntitiesByKey().putIfAbsent(
        key,
        holder
    );
    if (oldHolder != null) {
        assert oldHolder.entity == null || oldHolder.entity == entity;
        oldHolder.entity = entity;
        holder = oldHolder;
    }
    holder.state = EntityHolderState.INITIALIZED;
    final BatchFetchQueue fetchQueue = this.batchFetchQueue;
    if (fetchQueue != null) {
        fetchQueue.removeBatchLoadableEntityKey(key);
    }
}

SessionImpl is the Hibernate implementation
for EntityManager APIs, creates PersistenceContext
for each EntityManager and is doing first-level
caching in the map

```

The diagram illustrates the flow of persisting a User object. It starts with the `UserDetailsService` saving a user, which triggers the `save` method in `SimpleJpaRepository`. This method uses the `EntityManager` to either persist or merge the entity. The `SessionImpl` is highlighted as the Hibernate implementation for `EntityManager` APIs, handling first-level caching. Finally, the `UserDetails` object is returned.

During application startup (JPA creates fresh DB and table):

```

2024-11-11T15:00:46.273+05:30 INFO 9537 --- [           main] o.h.e.t.j.p.i.JtaPlatformInitiator      : HHH000489: No JTA platform available [set 'hibernate.transaction.jta.platform' to 'true' to enable JTA]
Hibernate:
  drop table if exists user_details cascade
Hibernate:
  create table user_details (
    id bigint generated by default as identity,
    email varchar(255),
    name varchar(255),
    primary key (id)
)
2024-11-11T15:00:46.285+05:30 INFO 9537 --- [           main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
2024-11-11T15:00:46.374+05:30  WARN 9537 --- [           main] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database queries may t
2024-11-11T15:00:46.555+05:30  INFO 9537 --- [           main] o.s.d.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8080 (http) with context path '/'
2024-11-11T15:00:46.560+05:30  INFO 9537 --- [           main] c.c.l.SpringbootApplication          : Started SpringbootApplication in 1.542 seconds (process running for 1.7)

```

When invoked /test-jpa API:

The screenshot shows a successful GET request to `/test-jpa` with a response body containing a User object:

```

{
  "id": 1,
  "name": "Key",
  "email": "key@conceptandcoding.com"
}

```

```

Hibernate:
    drop table if exists user_details cascade
Hibernate:
    create table user_details (
        id bigint generated by default as identity,
        email varchar(255),
        name varchar(255),
        primary key (id)
    )
2024-11-17T15:02:28.409+05:30 INFO 9564 --- [main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'spring-jpa'
2024-11-17T15:02:28.409+05:30 WARN 9564 --- [main] JpaBaseConfiguration$JpaWebConfiguration : spring-jpa.open-in-view is enabled by default. Therefore, all entities with an @Entity annotation will have a corresponding view entity generated on top of them!
2024-11-17T15:02:28.467+05:30 INFO 9564 --- [main] o.s.o.e.EmbeddedTomcatWebServer : Tomcat started on port 8080 (http) with context path '/'
2024-11-17T15:02:28.467+05:30 INFO 9564 --- [main] c.c.l.SpringbootApplication : Started SpringbootApplication in 1.404 seconds (process run time)
2024-11-17T15:02:32.097+05:30 INFO 9564 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet 'dispatcherServlet'
2024-11-17T15:02:32.097+05:30 INFO 9564 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
2024-11-17T15:02:32.097+05:30 INFO 9564 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 1 ms
Hibernate:
    insert
    into
        user_details
        (email, name, id)
    values
        (?, ?, default)

```

Only insert call, even though we are doing first insert and then read

invoked /read-jpa API

```

GET      localhost:8080/api/read-jpa
Params Authorization Headers (6) Body Scripts Settings
Query Params
Key Value
Expect 100
Key Value
Body Cookies Headers (5) Test Results
Pretty Row Preview Visualize JSON ↴
1: {
2:   "id": 1,
3:   "name": "xyz",
4:   "email": "xyz@conceptandcoding.com"
5: }

```

```

2024-11-17T15:02:28.409+05:30 INFO 9564 --- [main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'spring-jpa'
2024-11-17T15:02:28.409+05:30 WARN 9564 --- [main] JpaBaseConfiguration$JpaWebConfiguration : spring-jpa.open-in-view is enabled by default. Therefore, all entities with an @Entity annotation will have a corresponding view entity generated on top of them!
2024-11-17T15:02:28.467+05:30 INFO 9564 --- [main] o.s.o.e.EmbeddedTomcatWebServer : Tomcat started on port 8080 (http) with context path '/'
2024-11-17T15:02:28.467+05:30 INFO 9564 --- [main] c.c.l.SpringbootApplication : Started SpringbootApplication in 1.404 seconds (process run time)
2024-11-17T15:02:32.097+05:30 INFO 9564 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet 'dispatcherServlet'
2024-11-17T15:02:32.097+05:30 INFO 9564 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
2024-11-17T15:02:32.097+05:30 INFO 9564 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 1 ms
Hibernate:
    insert
    into
        user_details
        (email, name, id)
    values
        (?, ?, default)
Hibernate:
    select
        ud1_0.id,
        ud1_0.email,
        ud1_0.name
    from
        user_details ud1_0
    where
        ud1_0.id=?

```

So, PersistenceContext scope is associated with EntityManager:

```
@Service
public class UserDetailsService {

    @Autowired
    EntityManagerFactory entityManagerFactory;

    public UserDetails saveUser(UserDetails user) {

        EntityManager entityManager = entityManagerFactory.createEntityManager(); //session1 created
        entityManager.getTransaction().begin(); //transaction created
        entityManager.persist(user);
        entityManager.find(UserDetails.class, primaryKey: 1L);
        UserDetails output = entityManager.find(UserDetails.class, primaryKey: 1L);
        System.out.println("i am able to find the data, name is:" + output.getName());
        entityManager.getTransaction().commit(); //transaction committed
        entityManager.close(); // session1 closed

        EntityManager entityManager2 = entityManagerFactory.createEntityManager(); //session2 created
        entityManager2.getTransaction().begin(); //transaction created
        entityManager2.find(UserDetails.class, primaryKey: 1L);
        UserDetails output2 = entityManager2.find(UserDetails.class, primaryKey: 1L);
        System.out.println("Session2: i am able to find the data, name is:" + output2.getName());
        entityManager2.getTransaction().commit(); //transaction committed
        entityManager2.close(); // session1 closed

        return output2;
    }
}
```

Output:

```
2024-11-11T16:58:59.995+05:30 INFO 10877 --- [           main] c.h.e.t.j.p.i.JtaPlatformInitiator      : HHH000489: No JTA platform available
Hibernate:
  drop table if exists user_details cascade
Hibernate:
  create table user_details (
    id bigint generated by default as identity,
    email varchar(255),
    name varchar(255),
    primary key (id)
  )
2024-11-11T16:58:59.025+05:30 INFO 10877 --- [           main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory
2024-11-11T16:58:59.025+05:30 WARN 10877 --- [           main] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Add 'spring.open-in-view=false' to application.properties to disable.
2024-11-11T16:58:59.236+05:30 INFO 10877 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8080 (http) with 1 contexts
2024-11-11T16:58:59.242+05:30 INFO 10877 --- [           main] c.c.l.SpringbootApplication            : Started SpringbootApplication in 1.39 seconds
2024-11-11T16:59:01.873+05:30 INFO 10877 --- [nio-8080-exec-2] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring DispatcherServlet
2024-11-11T16:59:01.873+05:30 INFO 10877 --- [nio-8080-exec-2] o.s.web.servlet.DispatcherServlet       : Initializing Servlet 'dispatcherServlet'
2024-11-11T16:59:01.873+05:30 INFO 10877 --- [nio-8080-exec-2] o.s.web.servlet.DispatcherServlet       : Completed initialization in 0 ms
Hibernate:
  insert
  into
    user_details
    (email, name, id)
  values
    (?, ?, ?)
i am able to find the data, name is:xyz
Hibernate:
  select
    ud1_.id,
    ud1_.email,
    ud1_.name
  from
    user_details ud1_
  where
    ud1_.id=?
Session2: i am able to find the data, name is:xyz
```

And EntityManager is created for each HTTP Request, so different methods within same HTTP Request share the EntityManager

## DispatcherServlet Code snapshot:

```
if (mappedHandler.apply.RequestMapping(processedRequest, response)) {
    return;
}

// Actually invoke the handler...
sv = sv.handle(processedRequest, response, mappedHandler.getHandler());
}

if (entityManager.isConcurrentHandlingStarted()) {
    return;
}

public void preHandle(HttpServletRequest request) throws DatabaseAccessException {
    String key = getParticipateAttributeValue();
    EntityManagerManager syncManager = WebUtils.getSyncEntityManager(request);
    if (!syncManager.isConcurrentResult()) && applyEntityManagerBindingInterceptor(syncManager, key)) {
        return;
    }

    EntityManagerFactory emf = obtainEntityManagerFactory();
    if ((transaction) == null) transactionManager.beginTransaction(emf);
    // Do not modify the EntityManager just mark the request accordingly.
    Integer count = (Integer) request.getAttribute(key, WebRequest.SCOPED_REQUEST);
    int newCount = (count != null ? count + 1 : 1);
    request.setAttribute(key, newCount, WebRequest.SCOPED_REQUEST);
}

else {
    logger.debug("Opening JPA EntityManager in OpenEntityManagerInViewInterceptor");
    try {
        EntityManager em = createEntityManager();
        EntityManagerHolder emHolder = new EntityManagerHolder(em);
        TransactionSynchronizationManager.bindResource(em, emHolder);

        AsyncRequestInterceptor interceptor = new AsyncRequestInterceptor(emf, emHolder);
        syncManager.registerInitializableInterceptor(key, interceptor);
        syncManager.registerTerminableInterceptor(key, interceptor);
    }
    catch (PersistenceException ex) {
        throw new DatabaseAccessException("Could not create JPA EntityManager", ex);
    }
}
```

```
    @RestController
    @RequestMapping(value = "/api/")
    public class UserController {
        @Autowired
        UserDetailsService userDetailsService;

        @PostMapping(path = "/user/{id}")
        public UserDetails createUser(@PathVariable("id") String id,
                                      @RequestBody User user) {
            userDetailsService.createUser(id, user);
            return userDetailsService.findById(id);
        }
    }

    @Service
    public class UserDetailsService {
        @Autowired
        UserDetailsRepository userDetailsRepository;

        public void saveUser(User user) {
            userDetailsRepository.save(user);
        }

        public UserDetails findById(@PathVariable("id") Long primaryKey) {
            return userDetailsRepository.findById(primaryKey).get();
        }
    }
}
```

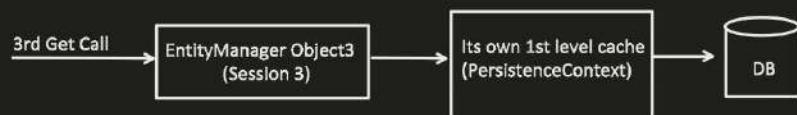
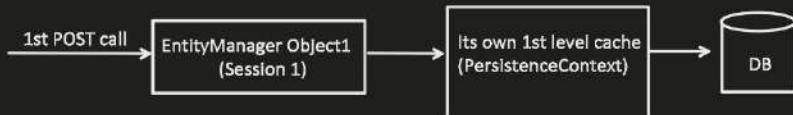
Internally both uses the same Entity Manager, therefore when `findById` Invoked, Cache Hit will happen

Internally both uses the same Entity Manager, therefore when `findById` Invoked, Cache Hit will happen

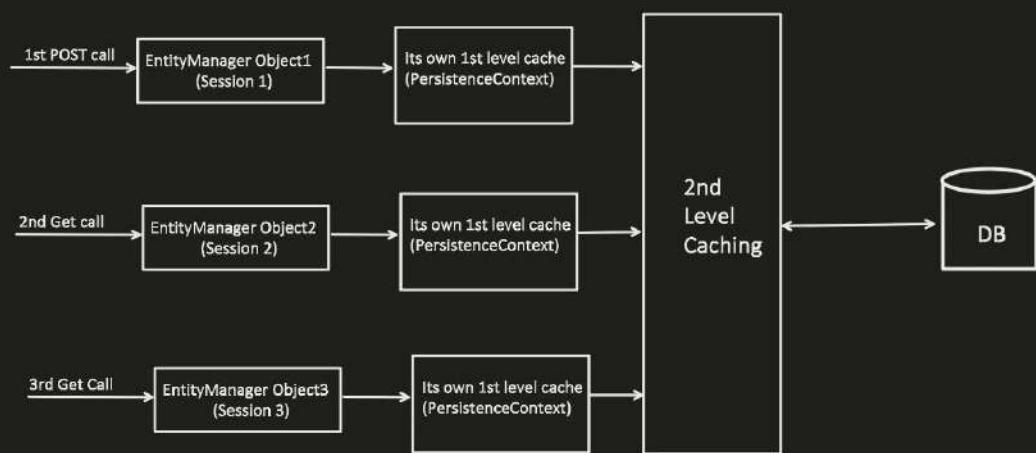
## Output:

## JPA (PART4)- Second Level Caching

From previous video, First level caching, we already know that, for each HTTP REQUEST, different EntityManager Object (session) is created and its have its own Persistence context (1st level cache)



Now, in **Second Level caching or L2 caching**, We will achieve something like this:



Lets first see, one happy flow, and see what all it takes to enable the 2nd level caching

pom.xml

```
<dependency>
    <groupId>org.ehcache</groupId>
    <artifactId>ehcache</artifactId>
    <version>3.16.8</version>
</dependency>
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-jcache</artifactId>
    <version>6.5.2.Final</version>
</dependency>
<dependency>
    <groupId>javax.cache</groupId>
    <artifactId>cache-api</artifactId>
    <version>1.1.1</version>
</dependency>
```

application.properties

```
spring.jpa.properties.hibernate.cache.use_second_level_cache=true
spring.jpa.properties.hibernate.cache.region.factory.class=org.hibernate.cache.jcache.JCacheRegionFactory
spring.jpa.properties.javax.cache.provider=org.ehcache.jsr107.EhcacheCachingProvider
logging.level.org.hibernate.cache.spi=DEBUG
```

```
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @Autowired
    UserDetailsService userDetailsService;

    @PostMapping(path = "/user")
    public UserDetails insertUser(@RequestBody UserDetails userDetails) {
        return userDetailsService.saveUser(userDetails);
    }

    @GetMapping("/user/{id}")
    public UserDetails getuser2D( {
        return userDetailsService.findById(primarykey, 1L);
    }
}

@Entity
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE,
       region = "UserDetailsCache")
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private String email;

    // Constructors
    public UserDetails() {
    }

    public UserDetails(String name, String email) {
        this.name = name;
        this.email = email;
    }

    // Getters and setters
}
```

```
@Service
public class UserDetailsService {

    @Autowired
    UserDetailsRepository userDetailsRepository;

    public UserDetails saveUser(UserDetails user) {
        return userDetailsRepository.save(user);
    }

    public UserDetails findById(Long primaryKey) {
        return userDetailsRepository.findById(primarykey).get();
    }
}

@Repository
public interface UserDetailsRepository extends
    JpaRepository<UserDetails, Long> {
}
```

1. During Insert, data is directly inserted into DB, no Cache insertion or validation happens

2. Get:

During Get, JPA will check, if data is present in cache? If Yes, Its cache hit and return else its cache miss and it will fetch from DB and put into Cache.

```
POST /userdetails
{
  "name": "Raj",
  "email": "raj@concentrandtesting.com"
}
```

```
GET /userdetails
{
  "name": "Raj",
  "email": "raj@concentrandtesting.com"
}

This is first Get call,
so cache Miss will happen and
DB call will be made
```

```
GET /userdetails
{
  "name": "Raj",
  "email": "raj@concentrandtesting.com"
}

Hey This Is Second Get call,
so cache hit will happen and
No DB call will be made
```

```
2024-11-25T20:43:25.766+05:30 INFO 7872 --- [nio-8080-exec-2] o.s.web.servlet.DispatcherServlet : Completed initialization in 1 ms
Hibernate:
    insert
    into
        user_details
        (email, name, id)
    values
        (?, ?, default)
2024-11-25T20:43:27.706+05:30 DEBUG 7872 --- [nio-8080-exec-3] o.h.c.s.support.AbstractReadWriteAccess : Getting cached data from region ['userDetailsCache']
2024-11-25T20:43:27.707+05:30 DEBUG 7872 --- [nio-8080-exec-3] o.h.c.s.support.AbstractReadWriteAccess : Cache miss : region = 'userDetailsCache'
Hibernate:
    select
        ud1_0.id,
        ud1_0.email,
        ud1_0.name
    from
        user_details ud1_0
    where
        ud1_0.id=?
2024-11-25T20:43:27.729+05:30 DEBUG 7872 --- [nio-8080-exec-3] o.h.c.s.support.AbstractReadWriteAccess : Caching data from load [region='userDetailsCache']
2024-11-25T20:43:28.292+05:30 DEBUG 7872 --- [nio-8080-exec-4] o.h.c.s.support.AbstractReadWriteAccess : Getting cached data from region ['userDetailsCache']
2024-11-25T20:43:28.293+05:30 DEBUG 7872 --- [nio-8080-exec-4] o.h.c.s.support.AbstractReadWriteAccess : Checking readability of read-write cache
2024-11-25T20:43:28.295+05:30 DEBUG 7872 --- [nio-8080-exec-4] o.h.c.s.support.AbstractReadWriteAccess : Cache hit : region = 'userDetailsCache',
'
```

### 1. Why in pom.xml, 3 dependencies required?

```
<dependency>
```

```
  <groupId>org.ehcache</groupId>
  <artifactId>ehcache</artifactId>
  <version>3.10.8</version>
</dependency>
```

Provides the core implementation of Second level caching

```
<dependency>
```

```
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-jcache</artifactId>
  <version>6.5.2.Final</version>
</dependency>
```

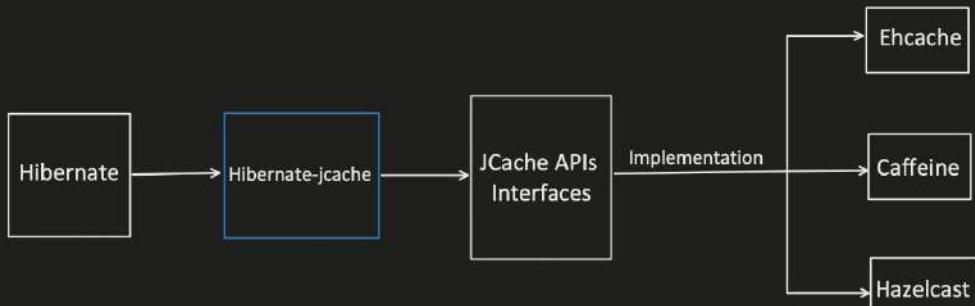
Hibernate specific Caching logic comes with this. Like we use Annotations over entity **@Cache**, we used **CacheConcurrencyStrategy**, so specific logic need to be executed, and this library help us with that.

```
<dependency>
```

```
  <groupId>javax.cache</groupId>
  <artifactId>cache-api</artifactId>
  <version>1.1.1</version>
</dependency>
```

Provides the interface for Jcache, hibernate interact with these APIs.

Helps to achieve Loose coupling. We can change from Ehcache to some other Jcache compliant caching provider without changing code.



## 2. Lets understand application.properties and Region:

*This tell hibernate to use hibernate-jcache class to manage caching, we can also provide here direct ehcache factory class, means bypassing Jcache interface*

```
spring.jpa.properties.hibernate.cache.use_second_level_cache=true  
spring.jpa.properties.hibernate.cache.region.factory_class=org.hibernate.cache.jcache.JCacheRegionFactory  
spring.jpa.properties.javax.cache.provider=org.ehcache.jsr107.EhcacheCachingProvider  
logging.level.org.hibernate.cache.spi=DEBUG
```

### Region:

Helps in logical grouping of cached data.

For each Region (or say group), we can apply different caching strategy like

- Eviction policy
- TTL
- Cache size
- Concurrency strategy etc.

Which helps in achieving granular level management of cached data (either Entity, Collection or Query results)

```

@Entity
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE,
      region = "userDetailsCache")
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private String email;

    // Constructors
    public UserDetails() {
    }

    public UserDetails(String name, String email) {
        this.name = name;
        this.email = email;
    }

    // Getters and setters
}

```

```

@Entity
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE,
      region = "orderDetailsCache")
public class OrderDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String productName;
    private int quantity;
    private double price;

    // Getters and Setters
}

```

**ehcache.xml**  
*(file within "src/main/resources/" path)*

```

<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:noNamespaceSchemaLocation="http://www.ehcache.org/ehcache.xsd">

    <cache alias="userDetailsCache"
          maxElementsInMemory="100"
          timeToLiveSeconds="60"
          evictionStrategy="LIFO" />

    <cache alias="orderDetailsCache"
          maxElementsInMemory="1000"
          timeToLiveSeconds="200"
          evictionStrategy="FIFO" />

```

### 3. Different *CacheConcurrencyStrategy*

```
@Entity  
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE,  
      region = "userDetailsCache")  
public class UserDetails {  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long id;  
  
    private String name;  
    private String email;  
  
    // Constructors  
    public UserDetails() {}  
  
    public UserDetails(String name, String email) {  
        this.name = name;  
        this.email = email;  
    }  
  
    // Getters and setters  
}
```

S.No.	Strategy
1.	READ_ONLY
2.	READ_WRITE
3.	NONSTRICT_READ_WRITE
4.	TRANSACTIONAL

#### 1. READ\_ONLY

- Good for Static Data
- Which do not require any updates
- If try to update just entity, exception will come

```

@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @Autowired
    UserDetailsService userDetailsService;

    @PostMapping(path = "/user")
    public UserDetails insertUser(@RequestBody UserDetails userDetails) {
        return userDetailsService.saveUser(userDetails);
    }

    @PutMapping(path = "/user/{id}")
    public UserDetails updateUser(@PathVariable Long id, @RequestBody UserDetails userDetails) {
        return userDetailsService.updateUser(id, userDetails);
    }

    @GetMapping("/user/{id}")
    public UserDetails getUser2() {
        return userDetailsService.findById(primarykey);
    }
}

@Entity
@Cache(usage = CacheConcurrencyStrategy.READ_ONLY,
      region = "userDetailsCache")
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private String email;

    // Constructors
    public UserDetails() {
    }

    public UserDetails(String name, String email) {
        this.name = name;
        this.email = email;
    }

    // Getters and setters
}

```

```

@Service
public class UserDetailsService {

    @Autowired
    UserDetailsRepository userDetailsRepository;

    public UserDetails saveUser(UserDetails user) {
        return userDetailsRepository.save(user);
    }

    public UserDetails updateUser(Long id, UserDetails user) {
        UserDetails existingUser = userDetailsRepository.findById(id).get();
        existingUser.setName(user.getName());
        existingUser.setEmail(user.getEmail());
        return userDetailsRepository.save(existingUser);
    }

    public UserDetails findById(Long primaryKey) {
        return userDetailsRepository.findById(primaryKey).get();
    }
}

@Repository
public interface UserDetailsRepository extends
    JpaRepository<UserDetails, Long> {
}

```

```

PUT      localhost:8080/api/user/1
Params   Authorization   Headers (8)   Body ●   Scripts   Settings
none   form-data   x-www-form-urlencoded   raw   binary   GraphQL   JSON
1  {
2    "name" : "sj_updated",
3    "email" : "xyz_updated@conceptandcoding.com"
4  }

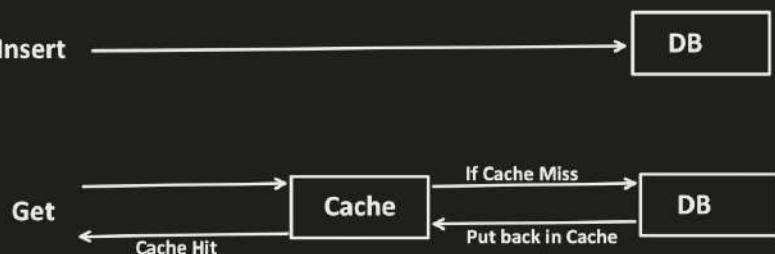
Body   Cookies   Headers (4)   Test Results   ⏱
Pretty   Raw   Preview   Visualize   JSON   ⚙️
1  {
2    "timestamp": "2024-12-14T11:25:55.758+00:00",
3    "status": 500,
4    "error": "Internal Server Error",
5    "path": "/api/user/1"
6  }

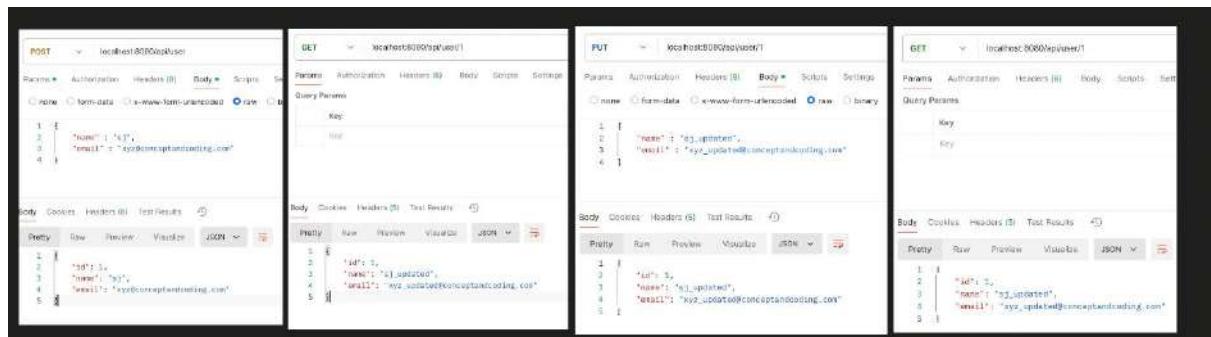
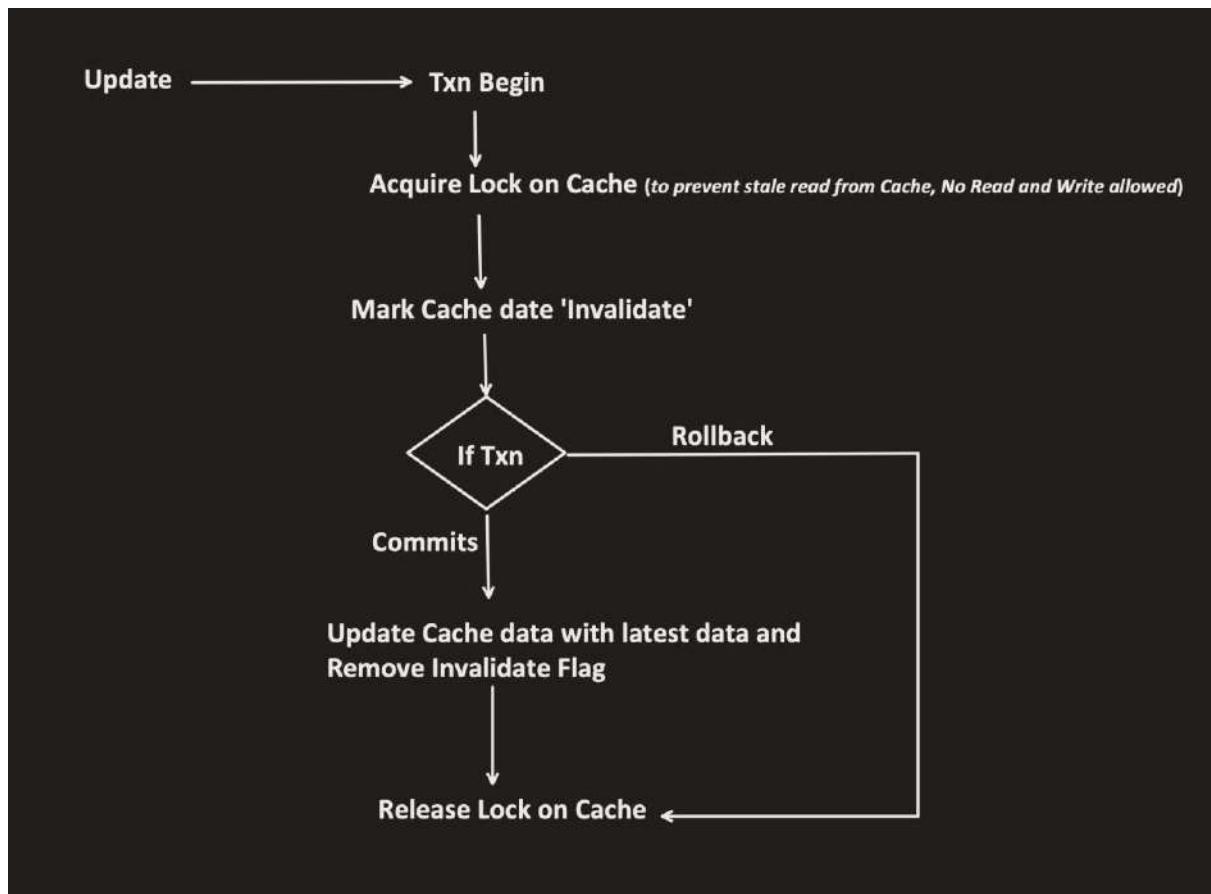
```

java.lang.UnsupportedOperationException: Create breakpoint! : Can't update readonly object  
at org.hibernate.cache.spi.support.EntityReadOnlyAccess.update(EntityReadOnlyAccess.java:71) ~[hibernate-core-6.5.2.Final.jar:6.5.2.Final]  
at org.hibernate.action.internal.EntityUpdateAction.updateCache(EntityUpdateAction.java:329) ~[hibernate-core-6.5.2.Final.jar:6.5.2.Final]  
at org.hibernate.action.internal.EntityUpdateAction.updateCacheItem(EntityUpdateAction.java:228) ~[hibernate-core-6.5.2.Final.jar:6.5.2.Final]

## 2. READ\_WRITE

- During Read, it put Shared Lock, other Read can also acquire Shared Lock. But no Write operation.
- During Update, it put Exclusive Lock, other Read and Write operation not allowed.





```

Hibernate:
    insert
        into
            user_details
            (email, name, id)
        values
            (?, ?, default)

→ Insert Operation (directly inserted into DB)

2024-03-14T20:16:29.059+05:30 DEBUG 33079 --- [nio-8888-exec-5] o.h.c.s.support.AbstractReadWriteAccess : Getting cached data from region ['userDetailsCache' (AccessType(read-write))] by key [cc
2024-03-14T20:16:29.060+05:30 DEBUG 33079 --- [nio-8888-exec-5] o.h.c.s.support.AbstractReadWriteAccess : Cache miss : region = 'userDetailsCache', key = 'cc.conceptandcoding.learningsprintboot
Hibernate:
    select
        ud0_.id,
        ud0_.name,
        ud0_.email
    from
        user_details ud0_
    where
        ud0_.id=?

→ Get Operation (Cache Miss, read from DB and inserted into Cache)

2024-03-14T20:16:29.061+05:30 DEBUG 33079 --- [nio-8888-exec-5] o.h.c.s.support.AbstractReadWriteAccess : Caching data from read [region='userDetailsCache' (AccessType(read-write))] : Key[cc
2024-03-14T20:16:29.061+05:30 DEBUG 33079 --- [nio-8888-exec-5] o.h.c.s.support.AbstractReadWriteAccess : Getting cached data from region ['userDetailsCache' (AccessType(read-write))] by key [cc
2024-03-14T20:16:29.061+05:30 DEBUG 33079 --- [nio-8888-exec-5] o.h.c.s.support.AbstractReadWriteAccess : Checking readability of read-write cache item [timestamp='71032325462784', version='n
2024-03-14T20:16:29.061+05:30 DEBUG 33079 --- [nio-8888-exec-5] o.h.c.s.support.AbstractReadWriteAccess : Cache hit : region = 'userDetailsCache', key = 'cc.conceptandcoding.learningsprintboot
2024-03-14T20:16:29.061+05:30 DEBUG 33079 --- [nio-8888-exec-5] o.h.c.s.support.AbstractReadWriteAccess : Locking cache item [region='userDetailsCache' (AccessType(read-write))] : 'cc.conceptandcodin
Hibernate:
    update
        user_details
    set
        name=?,
        name=?,
        where
        id=?

→ Update Operation (Cache Lock and update both DB and Cache on Success)

2024-03-14T20:17:56.476+05:30 DEBUG 33079 --- [nio-8888-exec-5] o.h.c.s.support.AbstractReadWriteAccess : Getting cached data from region ['userDetailsCache' (AccessType(read-write))] by key [cc
2024-03-14T20:17:56.477+05:30 DEBUG 33079 --- [nio-8888-exec-5] o.h.c.s.support.AbstractReadWriteAccess : Checking readability of read-write cache item [timestamp='7103232605873208', version='n
2024-03-14T20:17:56.478+05:30 DEBUG 33079 --- [nio-8888-exec-5] o.h.c.s.support.AbstractReadWriteAccess : Cache hit : region = 'userDetailsCache', key = 'cc.conceptandcoding.learningsprintboot

↓

Get Operation (Cache hit, No DB hit)

```

### 3. NONSTRICT\_READ\_WRITE

- During Read, No Lock is acquired at all.
- During Update, after txn commit successful, Cache is mark Invalidated and not updated with Fresh data.
- Good for Heavy Read application.
- So if Update and Read happens in parallel, its a chance that read operation get the stale data.

### 4. TRANSACTIONAL

- Acquire READ lock and Also WRITE lock.
- Updates the cache too, after txn commit successfully.
- Any other READ operation during cache lock, goes directly to DB.
- Any other WRITE operation during cache lock, waits in queue.

## JPA - PART5 (DTO - TABLE)

`spring.jpa.hibernate.ddl-auto` configuration tells hibernate regarding how to create and manage the DB Schema.

S.No.	Values	Create Schema	Update Schema	Delete Schema	Details
1.	none	no	no	no	Do nothing. Good for <b>Production</b>
2.	update	yes	yes	no	Does update but without deleting any existing data or schema. Good for <b>Development</b> environment
3.	validate	no	no	no	During application startup, does matching between entities and DB Schema. If mismatch found, throws exception.
4.	create	yes	yes	yes	Drops and re-create the schema during application startup.
5.	create-drop	Yes	yes	yes	Creates the schema during startup and drops the schema when the application shutdown. <i>(generally by-default for in memory databases like H2)</i>

### Mapping Classes to Tables

#### @Tables Annotation

- Its an Optional field, if not defined, hibernate will generate table name based on entity name.
- Generally it follows CamelCase to UPPER\_SNAKE\_CASE means: UserDetails -> USER\_DETAILS

```
@Target(TYPE)
@Retention(RUNTIME)
public @interface Table{
    String name() default "";
    String schema() default "";
    UniqueConstraint[] uniqueConstraints() default {};
    Index[] indexes() default {};
}
```

```

@Table(name= "USER_DETAILS")
@Entity
public class UserDetails {

//fields and their getters and setters
}

```

The screenshot shows a database browser interface. On the left, the schema tree is visible with nodes: jdbc:h2:mem:userDB, ORDER\_DETAILS, USER\_DETAILS, INFORMATION\_SCHEMA, and Users. A tooltip for the USER\_DETAILS node indicates it was last modified on H2 2.2.224 (2023-09-17). On the right, there are two tabs: 'Run Selected' and 'SQL statement'. The 'Run Selected' tab contains the SQL command 'SELECT \* FROM USER\_DETAILS;'. The results show three columns: ID, EMAIL, and NAME. Below the table, it says '(no rows, 3 ms)'. The 'SQL statement' tab also contains the same SQL command.

```

@Table(name= "USER_DETAILS", schema= "ONBOARDING")
@Entity
public class UserDetails {

//fields and their getters and setters
}

```

The screenshot shows a database browser interface. On the left, the schema tree is visible with nodes: jdbc:h2:mem:userDB, ORDER\_DETAILS, INFORMATION\_SCHEMA, ONBOARDING, USER\_DETAILS, and Users. A tooltip for the ONBOARDING node indicates it was last modified on H2 2.2.224 (2023-09-17). On the right, there are two tabs: 'Run Selected' and 'SQL statement'. The 'Run Selected' tab contains the SQL command 'SELECT \* FROM ONBOARDING.USER\_DETAILS;'. The results show three columns: ID, EMAIL, and NAME. Below the table, it says '(no rows, 3 ms)'. The 'SQL statement' tab also contains the same SQL command.

```

@Table(name= "USER_DETAILS",
       schema= "ONBOARDING",
       uniqueConstraints={
           @UniqueConstraint(columnNames="phone"), //single column unique constraint
           @UniqueConstraint(columnNames={"name","email"}) //composite unique constraint
       })
@Entity
public class UserDetails {

    @Id
    private Long id;
    private String name;
    private String email;
    private String phone;

    //Constructors
    public UserDetails(){}
    //Getters and setters
}

```

SELECT * FROM INFORMATION_SCHEMA.CONSTRAINT_COLUMN_USAGE;						
TABLE_CATALOG	TABLE_SCHEMA	TABLE_NAME	COLUMN_NAME	CONSTRAINT_CATALOG	CONSTRAINT_SCHEMA	CONSTRAINT_NAME
USERDB	ONBOARDING	USER_DETAILS	ID	USERDB	ONBOARDING	CONSTRAINT_3
USERDB	ONBOARDING	USER_DETAILS	PHONE	USERDB	ONBOARDING	UKHWI9ICWRBOMPOWJO1LBMQQU49
USERDB	ONBOARDING	USER_DETAILS	EMAIL	USERDB	ONBOARDING	UKDRYA4RWKEMVJ1HS8D8P7N59FM
USERDB	ONBOARDING	USER_DETAILS	NAME	USERDB	ONBOARDING	UKDRYA4RWKEMVJ1HS8D8P7N59FM

```

@Table(name= "USER_DETAILS",
       schema= "ONBOARDING",
       uniqueConstraints={

           @UniqueConstraint(columnNames="phone"), //single column unique constraint
           @UniqueConstraint(columnNames={"name","email"}) //composite unique constraint
       },
       indexes={

           @Index(name="index_phone", columnList="phone"), //index on single column
           @Index(name="index_name_email", columnList="name, email") //index on composite column
       })
@Entity
public class UserDetails {

    @Id
    private Long id;
    private String name;
    private String email;
    private String phone;

    //Constructors
    public UserDetails(){
    }

    //Getters and setters
}

```

SELECT \* FROM INFORMATION\_SCHEMA.INDEX\_COLUMNS

INDEX_CATALOG	INDEX_SCHEMA	INDEX_NAME	TABLE_CATALOG	TABLE_SCHEMA	TABLE_NAME	COLUMN_NAME
USERDB	ONBOARDING	PRIMARY_KEY_3	USERDB	ONBOARDING	USER_DETAILS	ID
USERDB	ONBOARDING	INDEX_PHONE	USERDB	ONBOARDING	USER_DETAILS	PHONE
USERDB	ONBOARDING	INDEX_NAME_EMAIL	USERDB	ONBOARDING	USER_DETAILS	NAME
USERDB	ONBOARDING	INDEX_NAME_EMAIL	USERDB	ONBOARDING	USER_DETAILS	EMAIL
USERDB	ONBOARDING	UKH1W194CWRBOMPOWJO1L8MQQU49_INDEX_3	USERDB	ONBOARDING	USER_DETAILS	PHONE
USERDB	ONBOARDING	UKDORYA4RNKEMVJ1HS8D8P7N59FM_INDEX_3	USERDB	ONBOARDING	USER_DETAILS	NAME
USERDB	ONBOARDING	UKDORYA4RNKEMVJ1HS8D8P7N59FM_INDEX_3	USERDB	ONBOARDING	USER_DETAILS	EMAIL

### @Column Annotation

- Its an Optional field, if not defined, JPA will add it with default values.

```

@Table(name = "user_details")
@Entity
public class UserDetails {

    @Id
    private Long id;
    @Column(name = "Full_name", unique = true, nullable = false, length = 255)
    private String name;
    private String email;
    private String phone;

    // Constructors
    public UserDetails() {
    }

    // Getters and setters
}

```

Run Selected Auto complete Clear SQL statement:

SELECT \* FROM USER\_DETAILS|

SELECT \* FROM USER\_DETAILS;

ID	EMAIL	FULL_NAME	PHONE
(no rows, 4 ms)			

### **@Id Annotation and @GeneratedValue Annotation**

**Primary Key:** must be unique, not null and used to uniquely identify each record.

- **@Id** annotation is used to mark the field as primary key.
- Each entity can have only 1 primary key.
- Only 1 field can be annotated with **@Id**.

**Composite Primary key:** combination of two or more columns to form a primary key.

```
graph TD; A(( )) --> B[Using @Embeddable and @EmbeddedId annotation.]; A --> C[Using @IdClass and @Id annotation]
```

### **Rules to follow for both the approach:**

- Must be a public class.
- Must Implement the Serializable interface.
- Must have no-arg constructor
- Must override the equals() and hashCode() methods

### Using `@IdClass` and `@Id` annotation

```
@Table(name = "user_details")
@Entity
public class UserDetails {

    private String name;           <-----> I want these 2 columns to be defined as Composite Key
    private String address;
    private String phone;

    // Constructors
    public UserDetails() {
    }

    // Getters and setters

}
```

```
@Table(name = "user_details")
@IdClass(UserDetailsCK.class)
@Entity
public class UserDetails {

    @Id
    private String name;
    @Id
    private String address;
    private String phone;

    // Constructors
    public UserDetails() {
    }

    // Getters and setters
}

public class UserDetailsCK implements Serializable {

    private String name;
    private String address;

    public UserDetailsCK() {
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (!(obj instanceof UserDetailsCK)) {
            return false;
        }
        UserDetailsCK userCK = (UserDetailsCK) obj;
        return this.name.equals(userCK.name) && this.address.equals(userCK.address);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, address);
    }
}
```

Run | Run Selected | Auto complete | Clear: SQL statement

SELECT \* FROM INFORMATION\_SCHEMA.INDEX\_COLUMNS

INDEX_CATALOG	INDEX_SCHEMA	INDEX_NAME	TABLE_CATALOG	TABLE_SCHEMA	TABLE_NAME	COLUMN_NAME	ORDINAL_POSITION	ORDERING_SPECIFICATION	NULL_ORDERING
USERDB	PUBLIC	PRIMARY_KEY_3	USERDB	PUBLIC	USER_DETAILS	ADDRESS	1	ASC	null
USERDB	PUBLIC	PRIMARY_KEY_3	USERDB	PUBLIC	USER_DETAILS	NAME	2	ASC	null

#### Why we need to override equals and hashCode methods?

- From previous video, we know that JPA internally maintains FIRST LEVEL CACHING.
- Also, we can implement SECOND LEVEL CACHING.
- And these caching uses HashMap and which relies on key (generally primary key becomes the key). So proper equals() and hashCode() method is required.

#### Why we need to implement Serializable interface?

- Unlike Single primary key like String, Long etc. Composite key are custom classes.
- So, JPA need to make sure that those class should be properly serializable (in case if required like transfer over the network in case of distributed caching).

Using `@Embeddable` and `@EmbeddedId` annotation.

```

@Table(name = "user_details")
@Entity
public class UserDetails {

    @EmbeddedId
    UserDetailsCK userDetailsService;
    private String phone;

    // Constructors
    public UserDetails() {
    }

    // Getters and setters
}

```

```

@Embeddable
public class UserDetailsCK implements Serializable {

    private String name;
    private String address;

    public UserDetailsCK() {
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (!(obj instanceof UserDetailsCK)) {
            return false;
        }
        UserDetailsCK userCK = (UserDetailsCK) obj;
        return this.name.equals(userCK.name) && this.address.equals(userCK.address);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, address);
    }
}

```

```

Run Run Selected Auto complete Clear SQL statement: SELECT * FROM USER_DETAILS;
SELECT * FROM INFORMATION_SCHEMA.INDEX_COLUMNS;
INDEX_CATALOG INDEX_SCHEMA INDEX_NAME TABLE_CATALOG TABLE_SCHEMA TABLE_NAME COLUMN_NAME ORDINAL_POSITION ORDERING_SPECIFICATION
USERDB PUBLIC PRIMARY_KEY_3_USERDB PUBLIC USER_DETAILS ADDRESS 1 ASC
USERDB PUBLIC PRIMARY_KEY_3_USERDB PUBLIC USER_DETAILS NAME 2 ASC
(2 rows, 2 ms)

Edit

```

### @GeneratedValue Annotation

- Now we know, how to define Primary key.  
- But we can also define its generation strategy too. By default, primary key columns are not autofill.  
- It works with @Id annotation (only for single primary key not for composite one)

1. GenerationType.IDENTITY

- Each insert, generates a new identifier (auto-increment field)

```

@GeneratedValue Annotation

@Entity
@Table(name = "user_details")
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String phone;

    // Constructors
    public UserDetails() {
    }

    public UserDetails(String name, String phone) {
        this.name = name;
        this.phone = phone;
    }

    // getters and setters
}

```

ID	NAME	PHONE
1	t1	1111
2	t2	1111

## 2. GenerationType.SEQUENCE

- Used to generate Unique numbers.
- Speed up the efficiency when we cache sequence values.
- More control than IDENTITY.

```
>> CREATE SEQUENCE user_seq INCREMENT BY 25 START WITH 100 MAXVALUE 9999;
```

```
@Table(name = "user_details")
@Entity
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "unique_user_seq")
    @SequenceGenerator(name = "unique_user_seq", sequenceName = "db_seq_name", initialValue = 100, allocationSize = 5)
    private Long id;
    private String name;
    private String phone;

    // Constructors
    public UserDetails() {
    }

    public UserDetails(String name, String phone) {
        this.name = name;
        this.phone = phone;
    }

    //getters and setters
}
```

The screenshot shows two Postman requests and a database query results section.

**Request 1 (Left):**

- Method: POST
- URL: localhost:8080/api/user
- Body (raw JSON):

```

1  {
2    "name": "ap",
3    "phone": "1111"
4  }

```

**Request 2 (Right):**

- Method: POST
- URL: localhost:8080/api/user
- Body (raw JSON):

```

1  {
2    "name": "bp",
3    "phone": "1111"
4  }

```

**Database Query Results:**

Run | Run Selected | Auto complete | Clear | SELECT \* FROM USER\_DETAILS;

SELECT \* FROM USER\_DETAILS;

ID	NAME	PHONE
100	ap	1111
101	bp	1111

(2 rows, 1 ms)

- **name**: it's a unique name, internal for JPA, we can use it in different entity.
- **sequenceName**: it's a name which is created in DB. Or if you have manually created the sequence in DB, then use that name here.
- **initialValue**: sequence no starts from.
- **allocationSize**: cache data, hibernate prefetch this much ids before hand, so that it will not query DB again and again.

We have given allocationSize = 5, so after 5 calls only, next db call will be made for sequence

```

Hibernate:
    insert
    into
        user_details
        (name, phone, id)
    values
        (?, ?, ?)
Hibernate:
    insert
    into
        user_details
        (name, phone, id)
    values
        (?, ?, ?)
Hibernate:
    insert
    into
        user_details
        (name, phone, id)
    values
        (?, ?, ?)
Hibernate:
    insert
    into
        user_details
        (name, phone, id)
    values
        (?, ?, ?)
Hibernate:
    insert
    into
        user_details
        (name, phone, id)
    values
        (?, ?, ?)
Hibernate:
    select
        next_value for db_seq_name
Hibernate:
    insert
    into
        user_details
        (name, phone, id)
    values
        (?, ?, ?)

```

→ **1st call**

→ **2nd call**

→ **3rd call**

→ **4th call**

→ **5th call**

→ **After 5th call, hibernate will fetch another 5 values.**

#### Advantage of SEQUENCE over IDENTITY:

1. Custom logic (start point, increment etc.)
2. Sequence generation logic is independent of table, so multiple tables can use it.
3. Range of IDs can be cached, so we can avoid hitting database each time a new id is required.  
(during IDENTITY, while INSERTION internally DB is auto generating the next ID, which require additional DB call)
4. Better portability, means IDENTITY is very DB specific while SEQUENCE can provide more consistent behavior across multiple DBs.

#### 3. GenerationType.TABLE

- @TableGenerator annotation is used but its very less efficient.
  - Because:
    - Separate Table is created, just for managing unique IDs.
    - Each time id is required, SELECT-UPDATE query is executed.
    - Complex concurrency handling, when multiple operations happening in parallel, it requires LOCK/UNLOCK functionality. Which can lead to performance bottleneck.
- In SEQUENCE type, its handle internally by DB using atomic counter, so its much more efficient.

## OneToOne Mapping (Unidirectional and Bidirectional)Part 6

### @OneToOne Unidirectional

- One Entity(A) references only one instance of another Entity(B).
- But reference exist only in one Direction i.e. from Parent(A) to Child(B).

```
@Table(name = "user_details")
@Entity
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String phone;

    @OneToOne(cascade = CascadeType.ALL)
    private UserAddress userAddress;

    // Constructors
    public UserDetails() {
    }

    //getters and setters
}
```

```
@Entity
@Table(name = "user_address")
public class UserAddress {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String street;
    private String city;
    private String state;
    private String country;
    private String pinCode;

    // Constructors
    public UserAddress() {
    }

}
```

ID	USER_ADDRESS_ID	NAME	PHONE

ID	CITY	COUNTRY	PIN_CODE	STATE	STREET

USER\_DETAILS

USER\_ADDRESS

user\_address\_id is a FK

- By default hibernate choose:
  - the Foreign Key (FK) name as : <field\_name\_id>  
that's why for "userAddress" it created the FK name as : user\_address\_id
  - Chooses the Primary Key (PK) of other table.

But If we need more control over it, we can use **@JoinColumn** annotation.

```
@Table(name = "user_details")
@Entity
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String phone;

    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "address_id", referencedColumnName = "id")
    private UserAddress userAddress;

    // Constructors
    public UserDetails() {
    }

    //getters and setters
}
```

```
@Entity
@Table(name = "user_address")
public class UserAddress {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String street;
    private String city;
    private String state;
    private String country;
    private String pinCode;

    // Constructors
    public UserAddress() {
    }

}
```

Run

Run Selected

Auto complete

Clear

SQL statement:

SELECT \* FROM USER\_DETAILS

SELECT \* FROM USER\_DETAILS;

ADDRESS_ID	ID	NAME	PHONE
------------	----	------	-------

(no rows, 5 ms)

What about Composite Key, how to reference it?

- We need to use *@JoinColumns* and need to map all columns.

```
@Embeddable  
public class UserAddressCK {  
  
    private String street;  
    private String pinCode;  
  
    //getters and setters  
}
```

```
@Entity  
@Table(name = "user_address")  
public class UserAddress {  
  
    @EmbeddedId  
    private UserAddressCK id;  
  
    private String city;  
    private String state;  
    private String country;  
  
    //getters and setters  
}
```

```

@Table(name = "user_details")
@Entity
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String phone;

    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumns({
        @JoinColumn(name = "address_street", referencedColumnName = "street"),
        @JoinColumn(name = "address_pin_code", referencedColumnName = "pinCode")
    })
    private UserAddress userAddress;

    // Constructors
    public UserDetails() {
    }

    //getters and setters
}

```

**Run**

**Run Selected**

**Auto complete**

**Clear**

**SQL statement:**

**SELECT \* FROM USER\_DETAILS**

**SELECT \* FROM USER\_DETAILS;**

ID	ADDRESS_PIN_CODE	ADDRESS_STREET	NAME	PHONE
----	------------------	----------------	------	-------

**(no rows, 2 ms)**

**Run** **Run Selected** **Auto complete** **Clear** SQL statement:

SELECT \* FROM USER\_ADDRESS

SELECT \* FROM USER\_ADDRESS;

**CITY** **COUNTRY** **PIN\_CODE** **STATE** **STREET**

(no rows, 1 ms)

SELECT * FROM INFORMATION_SCHEMA.INDEX_COLUMNS;								
INDEX_CATALOG	INDEX_SCHEMA	INDEX_NAME	TABLE_CATALOG	TABLE_SCHEMA	TABLE_NAME	COLUMN_NAME	ORDINAL_POSITION	ORDERING_SPECIFICATION
USERDB	PUBLIC	PRIMARY_KEY_3	USERDB	PUBLIC	USER_DETAILS	ID	1	ASC
USERDB	PUBLIC	CONSTRAINT_INDEX_3	USERDB	PUBLIC	USER_DETAILS	ADDRESS_PIN_CODE	1	ASC
USERDB	PUBLIC	CONSTRAINT_INDEX_3	USERDB	PUBLIC	USER_DETAILS	ADDRESS_STREET	2	ASC
USERDB	PUBLIC	PRIMARY_KEY_9	USERDB	PUBLIC	USER_ADDRESS	PIN_CODE	1	ASC
USERDB	PUBLIC	PRIMARY_KEY_9	USERDB	PUBLIC	USER_ADDRESS	STREET	2	ASC

**Now before we proceed with further Mappings, there is one more important thing i.e.**

#### **CascadeType**

**ALL**

**PERSIST**

**MERGE**

**REMOVE**

**REFRESH**

**DETACH**

- Without CascadeType, any operation on Parent do not affect Child entity.
- Managing Child entities explicitly can be error-prone.

#### **CascadeType.PERSIST**

- Persisting/Inserting the User entity automatically persists its associated UserAddress entity data.

```
@Entity
@Table(name = "user_address")
public class UserAddress {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String street;
    private String city;
    private String state;
    private String country;
    private String pinCode;

    //getters and setters
}
```

```
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @Autowired
    UserDetailsService userDetailsService;

    @PostMapping(path = "/user")
    public UserDetails insertUser(@RequestBody UserDetails userDetails) {
        return userDetailsService.saveUser(userDetails);
    }
}
```

```
@Service
public class UserDetailsService {

    @Autowired
    UserDetailsRepository userDetailsRepository;

    public UserDetails saveUser(UserDetails user) {
        return userDetailsRepository.save(user);
    }
}
```

```
@Repository
public interface UserDetailsRepository extends
    JpaRepository<UserDetails, Long> {
}
```

```
@Table(name = "user_details")
@Entity
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String phone;

    @OneToOne(cascade = CascadeType.PERSIST)
    @JoinColumn(name = "address_id", referencedColumnName = "id")
    private UserAddress userAddress;

    // Constructors
    public UserDetails() {
    }

    //getters and setters
}
```



localhost:8080/api/user

POST

localhost:8080/api/user

Params • Authorization Headers (8) Body • Scripts Settings

 none  form-data  x-www-form-urlencoded  raw  binary  GraphQL **JSON** ▾

```
1 {  
2   "name": "JohnXYZ",  
3   "phone": "1234567890",  
4   "userAddress": {  
5     "street": "123 Street",  
6     "city": "Bangalore",  
7     "state": "Karnataka",  
8     "country": "India",  
9     "pinCode": "10001"  
10    }  
11 }
```

Body Cookies Headers (5) Test Results ⏱

Pretty

Raw

Preview

Visualize

JSON ▾



```
1 {  
2   "id": 1,  
3   "name": "JohnXYZ",  
4   "phone": "1234567890",  
5   "userAddress": {  
6     "id": 1,  
7     "street": "123 Street",  
8     "city": "Bangalore",  
9     "state": "Karnataka",  
10    "country": "India",  
11    "pinCode": "10001"  
12  }  
13 }
```

**Run** **Run Selected** **Auto complete** **Clear** SQL statement:

```
SELECT * FROM USER_ADDRESS
```

```
SELECT * FROM USER_ADDRESS;
```

ID	CITY	COUNTRY	PIN_CODE	STATE	STREET
1	Bangalore	India	10001	Karnataka	123 Street

(1 row, 1 ms)

**Run** **Run Selected** **Auto complete** **Clear** SQL statement:

```
SELECT * FROM USER_DETAILS
```

```
SELECT * FROM USER_DETAILS;
```

ADDRESS_ID	ID	NAME	PHONE
1	1	JohnXYZ	1234567890

(1 row, 1 ms)

#### **CascadeType.MERGE**

- Updating the User entity automatically updates its associated UserAddress entity data.

Lets, first see, what happen if we use only **PERSIST** Cascade type

```
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @Autowired
    UserDetailsService userDetailsService;

    @PostMapping(path = "/user")
    public UserDetails insertUser(@RequestBody UserDetails userDetails) {
        return userDetailsService.saveUser(userDetails);
    }

    @PutMapping(path = "/user/{id}")
    public UserDetails updateUser(@PathVariable Long id, @RequestBody UserDetails userDetails) {
        return userDetailsService.updateUser(id, userDetails);
    }
}
```

```
@Service
public class UserDetailsService {

    @Autowired
    UserDetailsRepository userDetailsRepository;

    public UserDetails saveUser(UserDetails user) {
        return userDetailsRepository.save(user);
    }

    public UserDetails updateUser(Long id, UserDetails user) {
        Optional<UserDetails> existingUser = userDetailsRepository.findById(id);
        if(existingUser.isPresent()) {
            return userDetailsRepository.save(user);
        }
        return null;
    }
}
```

1st INSERT OPERATION



localhost:8080/api/user

POST



localhost:8080/api/user

Params

Authorization

Headers (8)

Body

Scripts

Settings

 none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
1  {
2      "name": "JohnXYZ",
3      "phone": "1234567890",
4      "userAddress": {
5          "street": "123 Street",
6          "city": "Bangalore",
7          "state": "Karnataka",
8          "country": "India",
9          "pinCode": "10001"
10     }
11 }
```

Body

Cookies

Headers (5)

Test Results



Pretty

Raw

Preview

Visualize

JSON



```
1  {
2      "id": 1,
3      "name": "JohnXYZ",
4      "phone": "1234567890",
5      "userAddress": {
6          "id": 1,
7          "street": "123 Street",
8          "city": "Bangalore",
9          "state": "Karnataka",
10         "country": "India",
11         "pinCode": "10001"
12     }
13 }
```

**Run**

**Run Selected**

**Auto complete**

**Clear**

**SQL statement:**

**SELECT \* FROM USER\_DETAILS**

**SELECT \* FROM USER\_DETAILS;**

ADDRESS_ID	ID	NAME	PHONE
1	1	JohnXYZ	1234567890

**(1 row, 3 ms)**

**Run**

**Run Selected**

**Auto complete**

**Clear**

**SQL statement:**

**SELECT \* FROM USER\_ADDRESS**

**SELECT \* FROM USER\_ADDRESS;**

ID	CITY	COUNTRY	PIN_CODE	STATE	STREET
1	Bangalore	India	10001	Karnataka	123 Street

**(1 row, 1 ms)**

**2nd UPDATE OPERATION:**

**(Changes made in both UserDetails and UserAddress, but only UserDetails changes reflected)**



localhost:8080/api/user/1

PUT



localhost:8080/api/user/1

Params

Authorization

Headers (8)

Body

Scripts

Settings

 none    form-data    x-www-form-urlencoded    raw    binary    GraphQL   **JSON**

```
1  {
2      "id": 1,
3      "name": "JohnXYZ_updated",
4      "phone": "1234567890",
5      "userAddress": {
6          "id": 1,
7          "street": "123 Street",
8          "city": "Bengaluru",
9          "state": "Karnataka",
10         "country": "India",
11         "pinCode": "10001"
12     }
13 }
```

Body

Cookies

Headers (5)

Test Results



Pretty

Raw

Preview

Visualize

JSON



```
1  {
2      "id": 1,
3      "name": "JohnXYZ_updated",
4      "phone": "1234567890",
5      "userAddress": {
6          "id": 1,
7          "street": "123 Street",
8          "city": "Bangalore",
9          "state": "Karnataka",
10         "country": "India",
11         "pinCode": "10001"
12     }
13 }
```

**Run** **Run Selected** **Auto complete** **Clear** SQL statement:

```
SELECT * FROM USER_DETAILS
```

```
SELECT * FROM USER_DETAILS;
```

ADDRESS_ID	ID	NAME	PHONE
1	1	JohnXYZ_updated	1234567890

(1 row, 1 ms)

**Run** **Run Selected** **Auto complete** **Clear** SQL statement:

```
SELECT * FROM USER_ADDRESS
```

```
SELECT * FROM USER_ADDRESS;
```

ID	CITY	COUNTRY	PIN_CODE	STATE	STREET
1	Bangalore	India	10001	Karnataka	123 Street

(1 row, 0 ms)

**Edit**

Now If I want to both INSERT and then UPDATE association capability, means I need both PERSIST + MERGE Cascade Type

```
@Table(name = "user_details")
@Entity
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String phone;

    @OneToOne(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
    @JoinColumn(name = "address_id", referencedColumnName = "id")
    private UserAddress userAddress;

    // Constructors
    public UserDetails() {
    }

    //getters and setters
}
```

1st INSERT OPERATION , Similar like above.



localhost:8080/api/user

POST



localhost:8080/api/user

Params

Authorization

Headers (8)

Body

Scripts

Settings

 none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
1  {
2      "name": "JohnXYZ",
3      "phone": "1234567890",
4      "userAddress": {
5          "street": "123 Street",
6          "city": "Bangalore",
7          "state": "Karnataka",
8          "country": "India",
9          "pinCode": "10001"
10     }
11 }
```

Body

Cookies

Headers (5)

Test Results



Pretty

Raw

Preview

Visualize

JSON



```
1  {
2      "id": 1,
3      "name": "JohnXYZ",
4      "phone": "1234567890",
5      "userAddress": {
6          "id": 1,
7          "street": "123 Street",
8          "city": "Bangalore",
9          "state": "Karnataka",
10         "country": "India",
11         "pinCode": "10001"
12     }
13 }
```

Run

Run Selected

Auto complete

Clear

SQL statement:

SELECT \* FROM USER\_DETAILS

SELECT \* FROM USER\_DETAILS;

ADDRESS_ID	ID	NAME	PHONE
1	1	JohnXYZ	1234567890

(1 row, 3 ms)

Run

Run Selected

Auto complete

Clear

SQL statement:

SELECT \* FROM USER\_ADDRESS

SELECT \* FROM USER\_ADDRESS;

ID	CITY	COUNTRY	PIN_CODE	STATE	STREET
1	Bangalore	India	10001	Karnataka	123 Street

(1 row, 1 ms)

2nd UPDATE OPERATION



localhost:8080/api/user/1

PUT



localhost:8080/api/user/1

Params

Authorization

Headers (8)

Body

Scripts

Settings

 none    form-data    x-www-form-urlencoded    raw    binary    GraphQL   **JSON**

```
1 {  
2   | "id": 1,  
3   | "name": "JohnXYZ_updated",  
4   | "phone": "1234567890",  
5   | "userAddress": {  
6     |   | "id": 1,  
7     |   | "street": "123 Street",  
8     |   | "city": "Bengaluru",  
9     |   | "state": "Karnataka",  
10    |   | "country": "India",  
11    |   | "pinCode": "10001"  
12  }  
13 }
```

Body

Cookies

Headers (5)

Test Results



Pretty

Raw

Preview

Visualize

JSON



```
1 {  
2   | "id": 1,  
3   | "name": "JohnXYZ_updated",  
4   | "phone": "1234567890",  
5   | "userAddress": {  
6     |   | "id": 1,  
7     |   | "street": "123 Street",  
8     |   | "city": "Bengaluru",  
9     |   | "state": "Karnataka",  
10    |   | "country": "India",  
11    |   | "pinCode": "10001"  
12  }  
13 }
```

**Run** **Run Selected** **Auto complete** **Clear** SQL statement:

```
SELECT * FROM USER_DETAILS |
```

```
SELECT * FROM USER_DETAILS;
```

ADDRESS_ID	ID	NAME	PHONE
1	1	JohnXYZ_updated	1234567890

(1 row. 1 ms)

**Run** **Run Selected** **Auto complete** **Clear** SQL statement:

```
SELECT * FROM USER_ADDRESS
```

```
SELECT * FROM USER_ADDRESS;
```

ID	CITY	COUNTRY	PIN_CODE	STATE	STREET
1	Bengaluru	India	10001	Karnataka	123 Street

(1 row, 2 ms)

We are using the same "save" method to update,  
internally it check, if its new entity or not, by looking  
for ID field present in entity object. Since we are passing  
ID in Request body, it will try to update instead of insert.

## CascadeType.REMOVE

- Deleting the User entity automatically delete its associated UserAddress entity data.

```
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @Autowired
    UserDetailsService userDetailsService;

    @PostMapping(path = "/user")
    public UserDetails insertUser(@RequestBody UserDetails userDetails) {
        return userDetailsService.saveUser(userDetails);
    }

    @PutMapping(path = "/user/{id}")
    public UserDetails updateUser(@PathVariable Long id, @RequestBody UserDetails userDetails) {
        return userDetailsService.updateUser(id, userDetails);
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteUser(@PathVariable Long id) {
        userDetailsService.deleteUser(id);
        return ResponseEntity.noContent().build();
    }
}
```

```

@Service
public class UserDetailsService {

    @Autowired
    UserDetailsRepository userDetailsRepository;

    public UserDetails saveUser(UserDetails user) {
        return userDetailsRepository.save(user);
    }

    public UserDetails updateUser(Long id, UserDetails user) {
        Optional<UserDetails> existingUser = userDetailsRepository.findById(id);
        if(existingUser.isPresent()) {
            return userDetailsRepository.save(user);
        }
        return null;
    }

    public void deleteUser(Long userId) {
        userDetailsRepository.deleteById(userId);
    }
}

```

```

@Table(name = "user_details")
@Entity
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String phone;

    @OneToOne(cascade = {CascadeType.PERSIST, CascadeType.MERGE, CascadeType.REMOVE})
    @JoinColumn(name = "address_id", referencedColumnName = "id")
    private UserAddress userAddress;

    // Constructors
    public UserDetails() {
    }

    //getters and setters
}

```

localhost:8080/api/user

POST localhost:8080/api/user

Params • Authorization Headers (8) Body • Scripts Settings

None form-data x-www-form-urlencoded raw binary GraphQL JSON

```
1 {  
2   "name": "JohnXYZ",  
3   "phone": "1234567890",  
4   "userAddress": {  
5     "street": "123 Street",  
6     "city": "Bangalore",  
7     "state": "Karnataka",  
8     "country": "India",  
9     "pinCode": "10001"  
10    }  
11 }
```

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

```
1 {  
2   "id": 1,  
3   "name": "JohnXYZ",  
4   "phone": "1234567890",  
5   "userAddress": {  
6     "id": 1,  
7     "street": "123 Street",  
8     "city": "Bangalore",  
9     "state": "Karnataka",  
10    "country": "India",  
11    "pinCode": "10001"  
12  }  
13 }
```

1st INSERT OPERATION , Similar like above.

2nd DELETE OPERATION

**DELETE** localhost:8080/api/user/1

Params Authorization Headers (6) Body Scripts Settings

Query Params

Key	Value	Description
Key	Value	Description

Body Cookies Headers (3) Test Results

Pretty Raw Preview Visualize Text

204 No Content

**Run Run Selected Auto complete Clear SQL statement:**

**SELECT \* FROM USER\_DETAILS**

**SELECT \* FROM USER\_DETAILS;**

ADDRESS_ID	ID	NAME	PHONE
------------	----	------	-------

**(no rows, 1 ms)**

**Run** **Run Selected** **Auto complete** **Clear** SQL statement:

SELECT \* FROM USER\_ADDRESS

SELECT \* FROM USER\_ADDRESS;

ID	CITY	COUNTRY	PIN_CODE	STATE	STREET
(no rows, 1 ms)					

#### **CascadeType.REFRESH and CascadeType.DETACH**

- Both are Less frequently used.

```
@Transactional
public <S extends T> S save(S entity) {
    Assert.notNull(entity, message: "Entity must not be null");
    if (this.entityInformation.isNew(entity)) {
        this.entityManager.persist(entity);
        return entity;
    } else {
        return this.entityManager.merge(entity);
    }
}
```

#### **CascadeType.REFRESH**

Internally JPA code, has access to EntityManager object, which maintains persistenceContext and thus handles FIRST LEVEL CACHING

```

@Service
public class UserDetailsService {

    @Autowired
    UserDetailsRepository userDetailsRepository;

    public UserDetails saveUser(UserDetails user) {
        return userDetailsRepository.save(user);
    }

    public UserDetails updateUser(Long id, UserDetails user) {
        Optional<UserDetails> existingUser = userDetailsRepository.findById(id);
        if(existingUser.isPresent()) {
            return userDetailsRepository.save(user);
        }
        return null;
    }

    public void deleteUser(Long userId) {
        userDetailsRepository.deleteById(userId);
    }
}

```

- Sometime, we want to BYPASS 'First Level Caching'
- So, EntityManager has one method called "refresh".
- What it does is, for that entity directly read the value from DB instead of First Level Cache.

```

/**
 * Refresh the state of the instance from the database,
 * overwriting changes made to the entity, if any.
 * @param entity entity instance
 * @throws IllegalArgumentException if the instance is not
 *         an entity or the entity is not managed
 * @throws TransactionRequiredException if there is no
 *         transaction when invoked on a container-managed
 *         entity manager of type <code>PersistenceContextType.TRANSACTION</code>
 * @throws EntityNotFoundException if the entity no longer
 *         exists in the database
 */
public void refresh(Object entity);

```

EntityManager.java

```

@Table(name = "user_details")
@Entity
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String phone;

    @OneToOne(cascade = {CascadeType.REFRESH})
    @JoinColumn(name = "address_id", referencedColumnName = "id")
    private UserAddress userAddress;

    // Constructors
    public UserDetails() {
    }

    //getters and setters
}

```

So, When we use the CascadeType.REFRESH, we tell JPA to not only read Parent Entity from DB but also its associated Child Entities too.

So in this case, whenever on USERDETAILS entity, entityManager will internally invoke 'refresh()' method', this Cascade type make sure that, USERADDRESS should also be read from DB not from First Level Cache.

CascadeType.DETACH

```

/**
 * Remove the given entity from the persistence context, causing
 * a managed entity to become detached. Unflushed changes made
 * to the entity if any (including removal of the entity),
 * will not be synchronized to the database. Entities which
 * previously referenced the detached entity will continue to
 * reference it.
 * @param entity entity instance
 * @throws IllegalArgumentException if the instance is not an
 *         entity
 * @since 2.0
 */
public void detach(Object entity);

```

- Similarly like persist, remove, refresh method. EntityManager also has 'detach' method.
- Which purpose is to remove the given entity from the PERSISTENCE CONTEXT.
- Means JPA is not managing its lifecycle now.

So, When we use the CascadeType.DETACH, we tell JPA to not only detach Parent Entity from persistence context but also its associated Child Entities too.

So in this case, whenever on USERDETAILS entity, entityManager will internally invoke 'detach()' method', this Cascade type make sure that, USERADDRESS should also be detached/removed from persistence context.

```

@Table(name = "user_details")
@Entity
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String phone;

    @OneToOne(cascade = {CascadeType.DETACH})
    @JoinColumn(name = "address_id", referencedColumnName = "id")
    private UserAddress userAddress;

    // Constructors
    public UserDetails() {
    }

    //getters and setters
}

```

#### **CascadeType.ALL**

- Means we want all the different Cascade types capabilities like PERSIST, MERGE, REMOVE, REFRESH & DETACH.

#### **Okay, we saw INSERT, UPDATE, REMOVE operation, but what about GET?**

Does child entities always get loaded with Parent Entity?

#### **Eager Loading**

#### **Lazy Loading**

- It means, associated entity is loaded immediately along with the parent entity.
- Default for @OneToOne and @ManyToOne
- It means, associated entity is NOT loaded immediately.
- Only loaded when explicitly accessed like when we call userDetail.getUserAddress().

- Default for @OneToMany, @ManyToMany.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)

public @interface OneToOne {

    /**
     * (Optional) The entity class that is the target of the association.
     * Defaults to the type of the field or property that stores the association.

    Class targetEntity() default void.class;

    /**
     * (Optional) The operations that must be cascaded to the target of the association.
     * By default no operations are cascaded.

    CascadeType[] cascade() default {};

    /**
     * (Optional) Whether the association should be lazily
     * loaded or must be eagerly fetched. The EAGER
     * strategy is a requirement on the persistence provider runtime that
     * the associated entity must be eagerly fetched. The LAZY
     * strategy is a hint to the persistence provider runtime.
     */
    FetchType fetch() default FetchType.EAGER;
}
```

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)

public @interface OneToMany {

    /**
     * (Optional) The entity class that is the target of the association. Optional only if the collection
     * property is defined using Java generics. Must be specified otherwise.
     *
     * Defaults to the parameterized type of the collection when defined using generics.

    Class targetEntity() default void.class;

    /**
     * (Optional) The operations that must be cascaded to the target of the association.
     *
     * Defaults to no operations being cascaded.
     *
     * When the target collection is a java.util.Map, the cascade element applies to the map value.

    CascadeType[] cascade() default {};

    /**
     * (Optional) Whether the association should be lazily loaded or must be eagerly fetched. The EAGER
     * strategy is a requirement on the persistence provider runtime that the associated entities must be
     * eagerly fetched. The LAZY strategy is a hint to the persistence provider runtime.

    FetchType fetch() default FetchType.LAZY;
}
```

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface ManyToMany {

    /**
     * (Optional) The entity class that is the target of the association. Optional only if the collection-valued
     * relationship property is defined using Java generics. Must be specified otherwise.
     *
     * Defaults to the parameterized type of the collection when defined using generics.

    Class targetEntity() default void.class;

    /**
     * (Optional) The operations that must be cascaded to the target of the association.
     *
     * When the target collection is a java.util.Map, the cascade element applies to the map value.
     *
     * Defaults to no operations being cascaded.

    CascadeType[] cascade() default {};

    /**
     * (Optional) Whether the association should be lazily loaded or must be eagerly fetched. The EAGER
     * strategy is a requirement on the persistence provider runtime that the associated entities must be
     * eagerly fetched. The LAZY strategy is a hint to the persistence provider runtime.

    FetchType fetch() default FetchType.LAZY;
}
```

```

@Target({METHOD, FIELD})
@Retention(RUNTIME)

public @interface ManyToOne {

    /**
     * (Optional) The entity class that is the target of the association.  

     * Defaults to the type of the field or property that stores the association.
     */
    Class<?> targetEntity() default void.class;

    /**
     * (Optional) The operations that must be cascaded to the target of the association.  

     * By default no operations are cascaded.
     */
    CascadeType[] cascade() default {};

    /**
     * (Optional) Whether the association should be lazily loaded or must be eagerly fetched. The EAGER  

     * strategy is a requirement on the persistence provider runtime that the associated entity must be  

     * eagerly fetched. The LAZY strategy is a hint to the persistence provider runtime.
     */
    FetchType fetch() default FetchType.EAGER;

    /**
     * (Optional) Whether the association is optional. If set to false then a non-null relationship must  

     * always exist.
     */
}

```

- JPA, do an assumption that, since there is only 1 child entity present, so possibly it might be required while accessing parent entity.
- But here, there can be many child entities present for a parent entity, so returning all the rows of the child entity might impact performance, so by-default it uses LAZY technique.

We can also control this default behavior:

```
@Table(name = "user_details")
@Entity
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String phone;

    @OneToOne(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
    @JoinColumn(name = "address_id", referencedColumnName = "id")
    private UserAddress userAddress;

    // Constructors
    public UserDetails() {
    }

    //getters and setters
}
```

```
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @Autowired
    UserDetailsService userDetailsService;

    @PostMapping(path = "/user")
    public UserDetails insertUser(@RequestBody UserDetails userDetails) {
        return userDetailsService.saveUser(userDetails);
    }

    @GetMapping("/user/{id}")
    public UserDetails fetchUser(@PathVariable Long id) {
        return userDetailsService.findById(id);
    }
}
```

```

@Service
public class UserDetailsService {

    @Autowired
    UserDetailsRepository userDetailsRepository;

    public UserDetails saveUser(UserDetails user) {
        return userDetailsRepository.save(user);
    }

    public UserDetails findByID(Long primaryKey) {
        return userDetailsRepository.findById(primaryKey).get();
    }
}

```

Lets try testing this code:

1st Insert Operation: (Success)

The screenshot shows a Postman request to `localhost:8080/api/user` using the `raw` JSON body:

```

1  {
2      "name": "JohnXYZ",
3      "phone": "1234567890",
4      "userAddress": {
5          "street": "123 Street",
6          "city": "Bangalore",
7          "state": "Karnataka",
8          "country": "India",
9          "pinCode": "10001"
10     }
11 }

```

The response is a `200 OK` with the same JSON data returned.

2nd Get Operation: (Failure)

The screenshot shows a REST API testing interface. At the top, a header bar indicates a GET request to the URL `localhost:8080/api/user/1`. Below this, a navigation bar includes tabs for Params, Authorization, Headers (6), Body, Scripts, and Settings. The Params tab is currently selected. Under the Params tab, there is a section titled "Query Params" with a table. The table has columns for Key, Value, and Description. There is one row in the table with the key "Key" and the value "Value". In the main content area, there are tabs for Body, Cookies, Headers (4), Test Results, and a refresh icon. The "Body" tab is selected. It contains sub-tabs for Pretty, Raw, Preview, Visualize, and JSON. The JSON tab is selected and displays a pretty-printed JSON error response. The response is as follows:

```

1
2   "timestamp": "2024-12-31T08:44:42.478+00:00",
3   "status": 500,
4   "error": "Internal Server Error",
5   "path": "/api/user/1"
6

```

In the top right corner of the main content area, the status code `500 Internal Server Error` is displayed.

GET operation failed because of loading UserAddress LAZILY, Since during JSON creation at the time of response, UserAddress data is missing thus library like JAKSON don't know how to serialize it and thus it failed while constructing the response.

```

com.fasterxml.jackson.databind.exc.InvalidDefinitionException Create breakpoint! : No serializer found for class org.hibernate.proxy.pojo.bytebuddy.ByteBuddyInterceptor
at com.fasterxml.jackson.databind.exc.InvalidDefinitionException.from(InvalidDefinitionException.java:77) ~[jackson-databind-2.17.2.jar:2.17.2]
at com.fasterxml.jackson.databind.SerializerProvider.reportBadDefinition(SerializerProvider.java:1330) ~[jackson-databind-2.17.2.jar:2.17.2]
at com.fasterxml.jackson.databind.DatabindContext.reportBadDefinition(DatabindContext.java:414) ~[jackson-databind-2.17.2.jar:2.17.2]

```

So, for GET operation, how to solve it:

- Use **@JsonIgnore**
- This will remove the UserAddress field totally for both Lazy and Eager loading.

```
@Table(name = "user_details")
@Entity
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String phone;

    @OneToOne(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
    @JoinColumn(name = "address_id", referencedColumnName = "id")
    @JsonIgnore
    private UserAddress userAddress;

    // Constructors
    public UserDetails() {
    }

    //getters and setters
}
```

GET localhost:8080/api/user/1

Params Authorization Headers (6) Body Scripts Settings

Query Params

	Key	Value
	Key	Value

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON 

```
1 {  
2   "id": 1,  
3   "name": "JohnXYZ",  
4   "phone": "1234567890"  
5 }
```

- Using **DTO (Data Transfer Object)**
- Much clean and recommended approach.
- Instead of sending Entity directly, first response will be mapped to our DTO object.

Hey, then how come INSERT operation, we don't see this serialization issue?

Its because during INSERT, we are inserting the data in DB and also its inserted into persistence Context (first level cache), so UserAddress data is already present in-memory. So when response is returned as part of same INSERT operation, it do not make any DB call, just fetched from the persistence context.

```
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @Autowired
    UserDetailsService userDetailsService;

    @PostMapping(path = "/user")
    public UserDetails insertUser(@RequestBody UserDetails userDetails) {
        return userDetailsService.saveUser(userDetails);
    }

    @GetMapping("/user/{id}")
    public UserDetailsDTO fetchUser(@PathVariable Long id) {
        return userDetailsService.findById(id).toDTO();
    }
}
```

```
@Service
public class UserDetailsService {

    @Autowired
    UserDetailsRepository userDetailsRepository;

    public UserDetails saveUser(UserDetails user) {
        return userDetailsRepository.save(user);
    }

    public UserDetails findById(Long primaryKey) {
        return userDetailsRepository.findById(primaryKey).get();
    }
}
```

```
@Table(name = "user_details")
@Entity
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String phone;

    @OneToOne(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
    @JoinColumn(name = "address_id", referencedColumnName = "id")
    private UserAddress userAddress;

    // Constructors
    public UserDetails() {
    }

    public UserDetailsDTO toDTO() {
        return new UserDetailsDTO(this);
    }

    //getters and setters
}
```

```
public class UserDetailsDTO {

    private Long id;
    private String name;
    private String phone;
    private String address;

    // Constructor to populate from UserDetails entity
    public UserDetailsDTO(UserDetails userDetails) {
        this.id = userDetails.getId();
        this.name = userDetails.getName();
        this.phone = userDetails.getPhone();
        System.out.println("going to query user address here now");
        this.address = userDetails.getUserAddress() != null ?
            userDetails.getUserAddress().getStreet() : null;
    }

    //getters and setters
}
```



localhost:8080/api/user/1

GET



localhost:8080/api/user/1

Params

Authorization

Headers (6)

Body

Scripts

Settings

## Query Params

	Key	Value
	Key	Value

Body

Cookies

Headers (5)

Test Results



Pretty

Raw

Preview

Visualize

JSON



```
1  {
2      "id": 1,
3      "name": "JohnXYZ",
4      "phone": "1234567890",
5      "address": "123 Street"
6 }
```

```
SpringbootApplication x

insert
into
    user_address
    (city, country, pin_code, state, street, id)
values
    (?, ?, ?, ?, ?, default)
Hibernate:
    insert
    into
        user_details
        (name, phone, address_id, id)
    values
        (?, ?, ?, default)
Hibernate:
    select
        ud1_0.id,
        ud1_0.name,
        ud1_0.phone,
        ud1_0.address_id
    from
        user_details ud1_0
    where
        ud1_0.id=?
going to query user address here now
Hibernate:
    select
        ua1_0.id,
        ua1_0.city,
        ua1_0.country,
        ua1_0.pin_code,
        ua1_0.state,
        ua1_0.street
    from
        user_address ua1_0
    where
        ua1_0.id=?
```

### Insert operation

During GET call, Because of LAZY,  
it did not fetched USERADDRESS

In DTO, before invoking getUserAddress, this getting printed.

Because of getUserAddress() call, DB select query hit happens

### @OneToOne bidirectional

- Both entities hold reference to each other, means:
  - UserDetails has a reference to UserAddress.
  - UserAddress also has a reference back to UserDetails (only in Object, not in DB table)

#### Owner side

#### Inverse side

- Holds the Foreign Key relationship

in a table.

- No Foreign key is created in table.
- Only holds **Object** reference of owing entity.

```
@Table(name = "user_details")
@Entity
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String phone;

    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "address_id", referencedColumnName = "id")
    private UserAddress userAddress;

    // Constructors
    public UserDetails() {
    }

    //getters and setters
}
```

```
@Entity
@Table(name = "user_address")
public class UserAddress {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String street;
    private String city;
    private String state;
    private String country;
    private String pinCode;

    @OneToOne(mappedBy = "userAddress", fetch = FetchType.EAGER)
    private UserDetails userDetails;

    //getters and setters
}
```

Run Run Selected Auto complete Clear SQL statement:

SELECT \* FROM USER\_DETAILS

SELECT \* FROM USER\_DETAILS;

ADDRESS_ID	ID	NAME	PHONE
------------	----	------	-------

(no rows, 5 ms)

**Run** **Run Selected** **Auto complete** **Clear** SQL statement:

**SELECT \* FROM USER\_ADDRESS**

**SELECT \* FROM USER\_ADDRESS;**

<b>ID</b>	<b>CITY</b>	<b>COUNTRY</b>	<b>PIN_CODE</b>	<b>STATE</b>	<b>STREET</b>
-----------	-------------	----------------	-----------------	--------------	---------------

(no rows, 4 ms)

Table looks exactly same as OneToOne Unidirectional only

address\_id is a FK

But we now have capability to go backward from UserAddress to UserDetails.

```
@RestController
@RequestMapping(value = "/api/")
public class UserAddressController {

    @Autowired
    UserAddressService userDetailsService;

    @GetMapping("/user-address/{id}")
    public UserAddress fetchUser(@PathVariable Long id) {
        return userDetailsService.findById(id);
    }
}
```

```
@Service
public class UserAddressService {

    @Autowired
    UserAddressRepository userAddressRepository;

    public UserAddress findByID(Long primaryKey) {
        return userAddressRepository.findById(primaryKey).get();
    }
}
```

Created a controller and service class to query UserAddress entity

Let's observe the behavior now,

1st: Insert Operation like before, using UserDetail controller API:

POST



localhost:8080/api/user

Params

Authorization

Headers (8)

Body

Scripts

Settings

none

form-data

x-www-form-urlencoded

raw

binary

GraphQL

**JSON**

```
1 {  
2   "name": "JohnXYZ",  
3   "phone": "1234567890",  
4   "userAddress": {  
5     "street": "123 Street",  
6     "city": "Bangalore",  
7     "state": "Karnataka",  
8     "country": "India",  
9     "pinCode": "10001"  
10    }  
11  }
```

Body

Cookies

Headers (5)

Test Results



Pretty

Raw

Preview

Visualize

JSON



```
1 {  
2   "id": 1,  
3   "name": "JohnXYZ",  
4   "phone": "1234567890",  
5   "userAddress": {  
6     "id": 1,  
7     "street": "123 Street",  
8     "city": "Bangalore",  
9     "state": "Karnataka",  
10    "country": "India",  
11    "pinCode": "10001",  
12    "userDetails": null  
13  }  
14 }
```

```

2024-12-31T16:45:04.749+05:30 INFO 39163 --- [nio-8080-exec-2] o.s.web.servlet.DispatcherServlet : Completed initialization in 1 ms
Hibernate:
    insert
    into
        user_address
        (city, country, pin_code, state, street, id)
    values
        (?, ?, ?, ?, ?, default)
Hibernate:
    insert
    into
        user_details
        (name, phone, address_id, id)
    values
        (?, ?, ?, default)

```

## 2nd: Get call of UserAddress Controller API:

localhost:8080/api/user-address/1

GET localhost:8080/api/user-address/1

Params Authorization Headers (6) Body Scripts Settings

Query Params

	Key	Value
	Key	Value

Body Cookies Headers (5) Test Results ⏪

Pretty Raw Preview Visualize JSON

```

18     "country": "India",
19     "pinCode": "10001",
20     "userDetails": {
21         "id": 1,
22         "name": "JohnXYZ",
23         "phone": "1234567890",
24         "userAddress": {
25             "id": 1,
26             "street": "123 Street",
27             "city": "Bangalore",
28             "state": "Karnataka",
29             "country": "India",
30             "pinCode": "10001",
31             "userDetails": {
32                 "id": 1,
33                 "name": "JohnXYZ",
34                 "phone": "1234567890",
35                 "userAddress": {
36                     "id": 1,
37                     "street": "123 Street",
38                     "city": "Bangalore",
39                     "state": "Karnataka",
40                     "country": "India",
41                     "pinCode": "10001",
42                     "userDetails": {
43                         "id": 1,
44                         "name": "JohnXYZ",
45                         "phone": "1234567890",
46                         "userAddress": {
47                             "id": 1,
48                             "street": "123 Street",
49                         }
50                     }
51                 }
52             }
53         }
54     }
55 }
```

```

select
    ua1_0.id,
    ua1_0.city,
    ua1_0.country,
    ua1_0.pin_code,
    ua1_0.state,
    ua1_0.street,
    ud1_0.id,
    ud1_0.name,
    ud1_0.phone
from
    user_address ua1_0
left join
    user_details ud1_0
    on ua1_0.id=ud1_0.address_id
where
    ua1_0.id=?
2024-12-31T16:47:23.393+05:30  WARN 39163 --- [nio-8088-exec-4] .w.s.m.s.DefaultHandlerExceptionResolver : Ignoring exception, response committed already: org.springframework.http.converter.HttpMessageNotWritableException:
2024-12-31T16:47:23.393+05:30  WARN 39163 --- [nio-8088-exec-4] .w.s.m.s.DefaultHandlerExceptionResolver : Resolved [org.springframework.http.converter.HttpMessageNotWritableException:

```

Successfully executed JOIN query from UserAddress to UserDetail table

But exception comes during Response building

#### INFINITE RECURSION

Why because :

During response construction:

1. Jackson starts serializing UserAddress after UserAddress is serialized.
2. It encounters UserDetails within UserAddress and starts serializing it.  
After UserDetails is serialized.
3. Inside UserDetails , it encounters UserAddress again and serialization keeps going on in loop.

Infinite Recursion issue in bidirectional mapping can be solved via:

[@JsonManagedReference and](#)

[@JsonBackReference](#)

- Should be Used only in Owning entity.
- Tells explicitly Jackson to go ahead and serialize the child entity.
- Should only be Used with Inverse/Child entity.
- Tells explicitly Jackson to not serialize the parent entity.

```
@Table(name = "user_details")
@Entity
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String phone;

    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "address_id", referencedColumnName = "id")
    @JsonManagedReference
    private UserAddress userAddress;

    // Constructors
    public UserDetails() {
    }

    //getters and setters
}
```

```
@Entity
@Table(name = "user_address")
public class UserAddress {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String street;
    private String city;
    private String state;
    private String country;
    private String pinCode;

    @OneToOne(mappedBy = "userAddress", fetch = FetchType.EAGER)
    @JsonBackReference
    private UserDetails userDetails;

    //getters and setters
}
```

@JsonManagedReference:

@JsonBackReference:



localhost:8080/api/user/1

GET



localhost:8080/api/user/1

Params

Authorization

Headers (6)

Body

Scripts

Settings

Query Params

	Key
	Key

Body

Cookies

Headers (5)

Test Results



Pretty

Raw

Preview

Visualize

JSON



```
1  {
2      "id": 1,
3      "name": "JohnXYZ",
4      "phone": "1234567890",
5      "userAddress": {
6          "id": 1,
7          "street": "123 Street",
8          "city": "Bangalore",
9          "state": "Karnataka",
10         "country": "India",
11         "pinCode": "10001"
12     }
13 }
```

GET API call on Parent Entity "UserDetail"

GET API call on Child Entity "UserAddress"



localhost:8080/api/user-address/1

GET

localhost:8080/api/user-address/1

Params

Authorization

Headers (6)

Body

Scripts

Setting:

### Query Params

	Key
	Key

Body

Cookies

Headers (5)

Test Results



Pretty

Raw

Preview

Visualize

JSON



```
1  {
2      "id": 1,
3      "street": "123 Street",
4      "city": "Bangalore",
5      "state": "Karnataka",
6      "country": "India",
7      "pinCode": "10001"
8  }
```

Is there a way that, I can load the associated entity from both side, but still avoid infinite recursion:

[@JsonIdentityInfo](#)

```
@Table(name = "user_details")
@Entity
@JsonIdentityInfo(
    generator = ObjectIdGenerators.PropertyGenerator.class,
    property = "id"
)
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String phone;

    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "address_id", referencedColumnName = "id")
    private UserAddress userAddress;

    // Constructors
    public UserDetails() {
    }

    //getters and setters
}
```

```

@Entity
@Table(name = "user_address")
@JsonIdentityInfo(
    generator = ObjectIdGenerators.PropertyGenerator.class,
    property = "id"
)
public class UserAddress {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String street;
    private String city;
    private String state;
    private String country;
    private String pinCode;

    @OneToOne(mappedBy = "userAddress", fetch = FetchType.EAGER)
    private UserDetails userDetails;

    //getters and setters

}

```

Any unique field from entity we can give, generally we give PK

- During serialization , Jackson gives the unique ID to the entity (based on property field).
- Though which Jackson can know, if that particular id entity is already serialized before, then it skip the serialization.

GET API call on Parent Entity "UserDetail"



localhost:8080/api/user/1

GET



localhost:8080/api/user/1

Params

Authorization

Headers (6)

Body

Scripts

Setting

Query Params

Key
Key

Body

Cookies

Headers (5)

Test Results



Pretty

Raw

Preview

Visualize

JSON



```
1  {
2      "id": 1,
3      "name": "JohnXYZ",
4      "phone": "1234567890",
5      "userAddress": {
6          "id": 1,
7          "street": "123 Street",
8          "city": "Bangalore",
9          "state": "Karnataka",
10         "country": "India",
11         "pinCode": "10001",
12         "userDetails": 1
13     }
14 }
```

GET API call on Child Entity "UserAddress"



localhost:8080/api/user-address/1

GET



localhost:8080/api/user-address/1

Params

Authorization

Headers (6)

Body

Scripts

Settings

Query Params

	Key
	Key

Body

Cookies

Headers (5)

Test Results



Pretty

Raw

Preview

Visualize

JSON



```
1  {
2      "id": 1,
3      "street": "123 Street",
4      "city": "Bangalore",
5      "state": "Karnataka",
6      "country": "India",
7      "pinCode": "10001",
8      "userDetails": {
9          "id": 1,
10         "name": "JohnXYZ",
11         "phone": "1234567890",
12         "userAddress": 1
13     }
14 }
```

## JPA-Part7

### One-to-Many:

- One entity associated with multiple records in another entity like: User can have many Orders.
- Reference exist only in 1 direction i.e. from Parent to Child.
- Since its 1:Many, means 1 parent have multiple child and we can not store multiple child ids in 1 parent row, so it creates a **NEW TABLE** and stores the mapping.
- By default its **Lazy** loading, means when query parents, child rows are not fetched.

### Unidirectional

```
@Table(name = "user_details")
@Entity
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String phone;

    @OneToMany(cascade = CascadeType.ALL)
    private List<OrderDetails> orderDetails = new ArrayList<>();

    // Constructors
    public UserDetails() {
    }

    //getters and setters
}
```

```
@Table(name = "order_details")
@Entity
public class OrderDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String productName;

    //getters and setters
}
```

Run

Run Selected Auto complete Clear SQL statement:

SELECT \* FROM USER\_DETAILS

SELECT \* FROM USER\_DETAILS;

ID	NAME	PHONE
----	------	-------

(no rows, 1 ms)

SQL statement:

SELECT \* FROM ORDER\_DETAILS

SELECT \* FROM ORDER\_DETAILS;

ID	PRODUCT_NAME
----	--------------

(no rows, 4 ms)

Run Run Selected Auto complete Clear SQL statement:

SELECT \* FROM USER\_DETAILS\_ORDER\_DETAILS |

SELECT \* FROM USER\_DETAILS\_ORDER\_DETAILS;

ORDER\_DETAILS\_ID    USER\_DETAILS\_ID

(no rows, 1 ms)

What if, we don't want to create new table:

- We can use `@JoinColumn`, this also tells JPA that we want to store the FK in Child table instead of creating a new table.

```
@Table(name = "order_details")
@Entity
public class OrderDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String productName;

    //getters and setters
}
```

```
@Table(name = "user_details")
@Entity
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long userId;
    private String name;
    private String phone;

    @OneToMany(cascade = CascadeType.ALL)
    @JoinColumn(name = "user_id_fk", referencedColumnName = "userId")
    private List<OrderDetails> orderDetails = new ArrayList<>();

    // Constructors
    public UserDetails() {
    }

    //getters and setters
}
```

Run Run Selected Auto complete Clear SQL statement:

SELECT \* FROM USER\_DETAILS

SELECT \* FROM USER\_DETAILS;

USER_ID	NAME	PHONE
---------	------	-------

(no rows, 1 ms)

Run

Run Selected

Auto complete

Clear

SQL statement:

SELECT \* FROM ORDER\_DETAILS

SELECT \* FROM ORDER\_DETAILS;

ID	USER_ID_FK	PRODUCT_NAME
----	------------	--------------

(no rows, 1 ms)



localhost:8080/api/user

POST



localhost:8080/api/user

Params • Authorization Headers (8) Body • Scripts Settings

 none  form-data  x-www-form-urlencoded  raw  binary  GraphQL **JSON**

```
1 {  
2   "name": "JohnXYZ",  
3   "phone": "1234567890",  
4   "orderDetails": [  
5     {  
6       "productName": "IceCream"  
7     },  
8     {  
9       "productName": "ColdDrinks"  
10    }  
11  ]  
12 }
```

Body Cookies Headers (5) Test Results

Pretty

Raw

Preview

Visualize

JSON



```
1 {  
2   "userId": 1,  
3   "name": "JohnXYZ",  
4   "phone": "1234567890",  
5   "orderDetails": [  
6     {  
7       "id": 1,  
8       "productName": "IceCream"  
9     },  
10    {  
11      "id": 2,  
12      "productName": "ColdDrinks"  
13    }  
14  ]  
15 }
```

**Run**

**Run Selected**

**Auto complete**

**Clear**

**SQL statement:**

**SELECT \* FROM USER\_DETAILS**

**SELECT \* FROM USER\_DETAILS;**

USER_ID	NAME	PHONE
1	JohnXYZ	1234567890

**(1 row, 1 ms)**

**Run**

**Run Selected**

**Auto complete**

**Clear**

**SQL statement:**

**SELECT \* FROM ORDER\_DETAILS**

**SELECT \* FROM ORDER\_DETAILS;**

ID	USER_ID_FK	PRODUCT_NAME
1	1	IceCream
2	1	ColdDrinks

**(2 rows, 1 ms)**

## 1. INSERT CALL

## 2. GET CALL (LAZY)

GET localhost:8080/api/user/1

Params Authorization Headers (6) Body Scripts Settings

Query Params

	Key	Value
	Key	Value

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

```
1 {
2   "id": 1,
3   "name": "JohnXYZ",
4   "phone": "1234567890",
5   "orders": [
6     {
7       "id": 1,
8       "productName": "IceCream"
9     },
10    {
11      "id": 2,
12      "productName": "ColdDrinks"
13    }
14  ]
15 }
```

```
Hibernate:  
    select  
        ud1_0.user_id,  
        ud1_0.name,  
        ud1_0.phone  
    from  
        user_details ud1_0  
    where  
        ud1_0.user_id=?  
going to map UserDetails to UserDTO  
going to query order table here now  
Hibernate:  
    select  
        od1_0.user_id_fk,  
        od1_0.id,  
        od1_0.product_name  
    from  
        order_details od1_0  
    where  
        od1_0.user_id_fk=?
```

Output:

```
public class UserDetailsDTO {

    private Long id;
    private String name;
    private String phone;
    private List<OrderDetails> orders;

    // Constructor to populate from UserDetails entity
    public UserDetailsDTO(UserDetails userDetails) {
        this.id = userDetails.getUserId();
        this.name = userDetails.getName();
        this.phone = userDetails.getPhone();
        System.out.println("going to query order table here now");
        this.orders = userDetails.getOrderDetails();
    }

    //getters and setters
}
```

```
@Table(name = "user_details")
@Entity
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long userId;
    private String name;
    private String phone;

    @OneToMany(cascade = CascadeType.ALL)
    @JoinColumn(name = "user_id_fk", referencedColumnName = "userId")
    private List<OrderDetails> orderDetails = new ArrayList<>();

    public UserDetailsDTO mapUserDetailsToUserDTO() {

        return new UserDetailsDTO(this);
    }

    //getters and setters

}
```

```

@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @Autowired
    UserDetailsService userDetailsService;

    @PostMapping(path = "/user")
    public UserDetails insertUser(@RequestBody UserDetails userDetails) {
        return userDetailsService.saveUser(userDetails);
    }

    @GetMapping("/user/{id}")
    public UserDetailsDTO fetchUser(@PathVariable Long id) {
        UserDetails output = userDetailsService.findById(id);
        System.out.println("going to map UserDetails to UserDTO");
        UserDetailsDTO userDTO = output.mapUserDetailsToUserDTO();
        return userDTO;
    }
}

```

EAGER fetch:

```

@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @Autowired
    UserDetailsService userDetailsService;

    @PostMapping(path = "/user")
    public UserDetails insertUser(@RequestBody UserDetails userDetails) {
        return userDetailsService.saveUser(userDetails);
    }

    @GetMapping("/user/{id}")
    public UserDetails fetchUser(@PathVariable Long id) {
        return userDetailsService.findById(id);
    }
}

```

```
@Table(name = "user_details")
@Entity
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long userId;
    private String name;
    private String phone;

    @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
    @JoinColumn(name = "user_id_fk", referencedColumnName = "userId")
    private List<OrderDetails> orderDetails = new ArrayList<>();

    // Constructors
    public UserDetails() {
    }

    //getters and setters

}
```

```

Hibernate:
    select
        ud1_0.user_id,
        ud1_0.name,
        ud1_0.phone,
        od1_0.user_id_fk,
        od1_0.id,
        od1_0.product_name
    from
        user_details ud1_0
    left join
        order_details od1_0
        on ud1_0.user_id=od1_0.user_id_fk
    where
        ud1_0.user_id=?

```

**Output:**

Cascade Type	Impact
CascadeType.PERSIST	Saving User(Parent) also saves related Orders (Child).
CascadeType.MERGE	Updating User also updates Orders.
CascadeType.REMOVE	Deleting User also deletes Orders.
CascadeType.ALL	Includes all cascade operations.

Different Cascade Types:

Orphan Removal:

Automatically removes child entry when child removed from Parent collection.

```
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @Autowired
    UserDetailsService userDetailsService;

    @PostMapping(path = "/user")
    public UserDetails insertUser(@RequestBody UserDetails userDetails) {
        return userDetailsService.saveUser(userDetails);
    }

    @GetMapping("/user/{id}")
    public UserDetails testOrphan(@PathVariable Long id) {
        UserDetails output = userDetailsService.findById(id);
        output.getOrderDetails().remove(index: 0);
        userDetailsService.saveUser(output);
        return output;
    }
}
```

1st: INSERT

HTTP [localhost:8080/api/user](http://localhost:8080/api/user)

POST [localhost:8080/api/user](#)

Params • Authorization Headers (8) Body • Scripts Settings

none  form-data  x-www-form-urlencoded  raw  binary  GraphQL **JSON** ▾

```
1 {  
2   "name": "JohnXYZ",  
3   "phone": "1234567890",  
4   "orderDetails": [  
5     {  
6       "productName": "IceCream"  
7     },  
8     {  
9       "productName": "ColdDrinks"  
10    }  
11  ]  
12 }
```

Body Cookies Headers (5) Test Results ⏪

Pretty Raw Preview Visualize **JSON** ▾

```
1 {  
2   "userId": 2,  
3   "name": "JohnXYZ",  
4   "phone": "1234567890",  
5   "orderDetails": [  
6     {  
7       "id": 3,  
8       "productName": "IceCream"  
9     },  
10    {  
11      "id": 4,  
12      "productName": "ColdDrinks"  
13    }  
14  ]  
15 }
```

Run Run Selected Auto complete Clear

SELECT \* FROM ORDER\_DETAILS |

SELECT \* FROM ORDER\_DETAILS;

ID	USER_ID_FK	PRODUCT_NAME
1	1	IceCream
2	1	ColdDrinks

(2 rows, 4 ms)

Run

Run Selected

Auto complete

Cle

SELECT \* FROM USER\_DETAILS

SELECT \* FROM USER\_DETAILS;

USER_ID	NAME	PHONE
1	JohnXYZ	1234567890

(1 row, 1 ms)

2nd: Testing Orphan Removal



localhost:8080/api/user/1

GET



localhost:8080/api/user/1

Params

Authorization

Headers (6)

Body

Scripts

Settings

### Query Params

	Key
	Key

Body

Cookies

Headers (5)

Test Results



Pretty

Raw

Preview

Visualize

JSON



```
1  {
2      "userId": 1,
3      "name": "JohnXYZ",
4      "phone": "1234567890",
5      "orderDetails": [
6          {
7              "id": 2,
8              "productName": "ColdDrinks"
9          }
10     ]
11 }
```

Run

Run Selected

Auto complete

Cle

SELECT \* FROM USER\_DETAILS

SELECT \* FROM USER\_DETAILS;

USER_ID	NAME	PHONE
1	JohnXYZ	1234567890

(1 row, 1 ms)

(With OrphanRemoval = false)

```
@Table(name = "user_details")
@Entity
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long userId;
    private String name;
    private String phone;

    @OneToMany(cascade = CascadeType.ALL, orphanRemoval = false)
    @JoinColumn(name = "user_id_fk", referencedColumnName = "userId")
    private List<OrderDetails> orderDetails = new ArrayList<>();

    // Constructors
    public UserDetails() {
    }

    //getters and setters
}
```

```
@Table(name = "user_details")
@Entity
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long userId;
    private String name;
    private String phone;

    @OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
    @JoinColumn(name = "user_id_fk", referencedColumnName = "userId")
    private List<OrderDetails> orderDetails = new ArrayList<>();

    // Constructors
    public UserDetails() {
    }

    //getters and setters
}
```



localhost:8080/api/user/1

GET



localhost:8080/api/user/1

Params

Authorization

Headers (6)

Body

Scripts

Settings

### Query Params

	Key
	Key

Body

Cookies

Headers (5)

Test Results



Pretty

Raw

Preview

Visualize

JSON



```
1  {
2      "userId": 1,
3      "name": "JohnXYZ",
4      "phone": "1234567890",
5      "orderDetails": [
6          {
7              "id": 2,
8              "productName": "ColdDrinks"
9          }
10     ]
11 }
```

Run

Run Selected

Auto complete

Cle

SELECT \* FROM USER\_DETAILS

SELECT \* FROM USER\_DETAILS;

USER_ID	NAME	PHONE
1	JohnXYZ	1234567890

(1 row, 1 ms)

Run

Run Selected

Auto complete

Clear

SQL statement:

SELECT \* FROM ORDER\_DETAILS

SELECT \* FROM ORDER\_DETAILS;

ID	USER_ID_FK	PRODUCT_NAME
2	1	ColdDrinks

(1 row, 1 ms)

```
Hibernate:  
    select  
        ud1_0.user_id,  
        ud1_0.name,  
        ud1_0.phone  
    from  
        user_details ud1_0  
    where  
        ud1_0.user_id=?  
  
Hibernate:  
    select  
        od1_0.user_id_fk,  
        od1_0.id,  
        od1_0.product_name  
    from  
        order_details od1_0  
    where  
        od1_0.user_id_fk=?  
  
Hibernate:  
    update  
        order_details  
    set  
        user_id_fk=null  
    where  
        user_id_fk=?  
        and id=?
```

**Console Output: Persistence Context knows all the info, which is present in memory**

```
Hibernate:
    select
        ud1_0.user_id,
        ud1_0.name,
        ud1_0.phone
    from
        user_details ud1_0
    where
        ud1_0.user_id=?
```

```
Hibernate:
    select
        od1_0.user_id_fk,
        od1_0.id,
        od1_0.product_name
    from
        order_details od1_0
    where
        od1_0.user_id_fk=?
```

```
Hibernate:
    update
        order_details
    set
        user_id_fk=null
    where
        user_id_fk=?
        and id=?
```

```
Hibernate:
    delete
    from
        order_details
    where
        id=?
```

**One-to-Many:**

**Bidirectional**

- Parent reference to child.
- Each Child reference to Parent.

Owning Side

Inverse Side

- Holds the Foreign Key relationship

in a table.

- No Foreign key is created in table.
- Only holds **Object** reference of owing entity.

```
@Table(name = "order_details")
@Entity
@JsonIdentityInfo(
    generator = ObjectIdGenerators.PropertyGenerator.class,
    property = "id")
public class OrderDetails {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String productName;
    @ManyToOne
    @JoinColumn(name = "user_id_owing_fk", referencedColumnName = "userID")
    private UserDetails userDetails;

    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public String getProductName() {
        return productName;
    }
    public void setProductName(String productName) {
        this.productName = productName;
    }
    public UserDetails getUserDetails() {
        return userDetails;
    }
    public void setUserDetails(UserDetails userDetails) {
        this.userDetails = userDetails;
    }
}
```

```
@Table(name = "user_details")
@Entity
@JsonIdentityInfo(
    generator = ObjectIdGenerators.PropertyGenerator.class,
    property = "userId")
public class UserDetails {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long userId;
    private String name;
    private String phone;
    @OneToMany(mappedBy = "userDetails", cascade = CascadeType.ALL)
    private List<OrderDetails> orderDetails = new ArrayList<>();

    public Long getUserId() {
        return userId;
    }
    public void setUserId(Long userId) {
        this.userId = userId;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getPhone() {
        return phone;
    }
    public void setPhone(String phone) {
        this.phone = phone;
    }
    public List<OrderDetails> getOrderDetails() {
        return orderDetails;
    }
    public void setOrderDetails(List<OrderDetails> orderDetails) {
        this.orderDetails = orderDetails;
        for(OrderDetails order: orderDetails) {
            order.setUserDetails(this);
        }
    }
}
```



localhost:8080/api/user

POST



localhost:8080/api/user

Params

Authorization

Headers (8)

Body

Scripts

Settings

 none     form-data     x-www-form-urlencoded     raw     binary    

```
1 {  
2   "name": "JohnXYZ",  
3   "phone": "1234567890",  
4   "orderDetails": [  
5     {  
6       "productName": "IceCream"  
7     },  
8     {  
9       "productName": "ColdDrinks"  
10    }  
11  ]  
12 }
```

Body

Cookies

Headers (5)

Test Results



Pretty

Raw

Preview

Visualize

JSON



```
1 {  
2   "userId": 1,  
3   "name": "JohnXYZ",  
4   "phone": "1234567890",  
5   "orderDetails": [  
6     {  
7       "id": 1,  
8       "productName": "IceCream",  
9       "userDetails": 1  
10      },  
11      {  
12        "id": 2,  
13        "productName": "ColdDrinks",  
14        "userDetails": 1  
15      }  
16    ]  
17 }
```

**Run**

**Run Selected**

**Auto complete**

**Clear**

**SQL statement:**

**SELECT \* FROM ORDER\_DETAILS**

**SELECT \* FROM ORDER\_DETAILS;**

ID	USER_ID_OWING_FK	PRODUCT_NAME
1	1	IceCream
2	1	ColdDrinks

**(2 rows, 1 ms)**

Run

Run Selected

Auto complete

Clear

SQL statement:

```
SELECT * FROM USER_DETAILS |
```

```
SELECT * FROM USER_DETAILS;
```

USER_ID	NAME	PHONE
1	JohnXYZ	1234567890



localhost:8080/api/user/1

GET



localhost:8080/api/user/1

Params

Authorization

Headers (6)

Body

Scripts

Settings

Query Params

	Key
	Key

Body

Cookies

Headers (5)

Test Results



Pretty

Raw

Preview

Visualize

JSON



```
1  {
2      "userId": 1,
3      "name": "JohnXYZ",
4      "phone": "1234567890",
5      "orderDetails": [
6          {
7              "id": 1,
8              "productName": "IceCream",
9              "userDetails": 1
10         },
11         {
12             "id": 2,
13             "productName": "ColdDrinks",
14             "userDetails": 1
15         }
16     ]
17 }
```

Many-to-One:

Unidirectional

- In this, we generally talk from the perspective of Child like : Many Orders can be placed by 1 User.
- So still, User is considered as Parent and Orders as Child.
- Parent don't have reference to child.
- Each child has reference to parent.

```
@Table(name = "user_details")
@Entity
public class UserDetails {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long userId;
    private String name;
    private String phone;

    //getters and setters
}
```

```
@Table(name = "order_details")
@Entity
public class OrderDetails {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String productName;
    @ManyToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "user_id_owing_fk", referencedColumnName = "userID")
    private UserDetails userDetails;

    //getter and setter
}
```

POST localhost:8080/api/order

Params • Authorization Headers (8) Body • Scripts Settings

none  form-data  x-www-form-urlencoded  raw  binary  GraphQL **JSON** ▾

```
1 {  
2   "productName": "IceCream",  
3   "userDetails":  
4     {  
5       "name": "SJ",  
6       "phone": "111212121221"  
7     }  
8 }
```

Body Cookies Headers (5) Test Results ↻

Pretty Raw Preview Visualize JSON ▾

```
1 {  
2   "id": 1,  
3   "productName": "IceCream",  
4   "userDetails": {  
5     "userId": 1,  
6     "name": "SJ",  
7     "phone": "111212121221"  
8   }  
9 }
```

```
@RestController
@RequestMapping(value = "/api/")
public class OrderController {

    @Autowired
    OrderService orderService;

    @GetMapping("/order/{id}")
    public OrderDetails fetchUser(@PathVariable Long id) {
        return orderService.findById(id);
    }

    @PostMapping(path = "/order")
    public OrderDetails insertOrder(@RequestBody OrderDetails orderDetails) {
        return orderService.saveOrder(orderDetails);
    }
}
```

```
@Service
public class OrderService {

    @Autowired
    OrderDetailsRepository orderDetailsRepository;

    public OrderDetails findById(Long primaryKey) {
        return orderDetailsRepository.findById(primaryKey).get();
    }

    public OrderDetails saveOrder(OrderDetails order) {
        return orderDetailsRepository.save(order);
    }
}
```

**Run** **Run Selected** **Auto complete** **Clear** SQL statement:

```
SELECT * FROM USER_DETAILS
```

```
SELECT * FROM USER_DETAILS;
```

USER_ID	NAME	PHONE
1	SJ	111212121221

(1 row, 1 ms)

**Run** **Run Selected** **Auto complete** **Clear** SQL statement:

```
SELECT * FROM ORDER_DETAILS |
```

```
SELECT * FROM ORDER_DETAILS;
```

ID	USER_ID_OWING_FK	PRODUCT_NAME
1	1	IceCream

(1 row, 1 ms)

Try it out:

Many-to-One: Bidirectional would be same as OneToMany Bidirectional

## **Many-to-Many: Unidirectional**

- Reference from One way only

```
@Table(name = "order_details")
@Entity
public class OrderDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long orderNo;

    @ManyToMany (cascade = CascadeType.ALL)
    @JoinTable(
        name = "order_product", // new Join table name
        joinColumns = @JoinColumn(name = "order_id"), // Foreign key for Order
        inverseJoinColumns = @JoinColumn(name = "product_id") // Foreign key for Product
    )
    private List<ProductDetails> productDetails = new ArrayList<>();

    public Long getOrderNo() {
        return orderNo;
    }

    public void setOrderNo(Long orderNo) {
        this.orderNo = orderNo;
    }

    public List<ProductDetails> getProductDetails() {
        return productDetails;
    }

    public void setProductDetails(List<ProductDetails> productDetails) {
        this.productDetails = productDetails;
    }
}
```

```
@Table(name = "product_details")
@Entity
public class ProductDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long productId;

    private String name;
    private double price;

    //getters and setters
}
```

Run

Run Selected

Auto complete

Clear

SQL statement:

SELECT \* FROM ORDER\_PRODUCT

SELECT \* FROM ORDER\_PRODUCT;

ORDER_ID	PRODUCT_ID
----------	------------

(no rows, 1 ms)

**Run** **Run Selected** **Auto complete** **Clear** SQL statement:

```
SELECT * FROM PRODUCT_DETAILS
```

```
SELECT * FROM PRODUCT_DETAILS;
```

PRICE	PRODUCT_ID	NAME
-------	------------	------

(no rows, 0 ms)

**Run** **Run Selected** **Auto complete** **Clear** SQL statement:

```
SELECT * FROM ORDER_DETAILS |
```

```
SELECT * FROM ORDER_DETAILS;
```

ORDER_NO
----------

(no rows, 1 ms)

```
@RestController
@RequestMapping(value = "/api/")
public class ProductController {

    @Autowired
    ProductService productService;

    @PostMapping(path = "/product")
    public ProductDetails insertUser(@RequestBody ProductDetails product) {
        return productService.saveProduct(product);
    }
}
```

```
@RestController
@RequestMapping(value = "/api/")
public class OrderController {

    @Autowired
    OrderService orderService;

    @Autowired
    ProductService productService;

    @GetMapping("/order/{id}")
    public OrderDetails fetchUser(@PathVariable Long id) {
        return orderService.findById(id);
    }

    @PostMapping(path = "/order")
    public OrderDetails insertOrder(@RequestBody OrderDetails orderDetail) {

        List<ProductDetails> managedProducts = orderDetail.getProductDetails().stream()
            .map(product -> productService.findById(product.getProductId()))
            .collect(Collectors.toList());

        orderDetail.setProductDetails(managedProducts);
        return orderService.saveOrder(orderDetail);
    }
}
```

1st : CREATE PRODUCTS

POST

localhost:8080/api/product

Params • Authorization Headers (8) Body • Scripts Settings

none  form-data  x-www-form-urlencoded  raw  binary

```
1  {
2    "name" : "ice-cream",
3    "price" : "100"
4 }
```

Body Cookies Headers (5) Test Results

Pretty

Raw

Preview

Visualize

JSON ▾



```
1  {
2    "productId": 1,
3    "name": "ice-cream",
4    "price": 100.0
5 }
```

POST



localhost:8080/api/product

Params •

Authorization

Headers (8)

Body •

Scripts

Settings

none    form-data    x-www-form-urlencoded    raw    binary    Graph

```
1  {
2    "name" : "cold-crink",
3    "price" : "150"
4 }
```

Body   Cookies   Headers (5)   Test Results

Pretty

Raw

Preview

Visualize

JSON



```
1  {
2    "productId": 2,
3    "name": "cold-crink",
4    "price": 150.0
5 }
```

Run Run Selected Auto complete Clear SQL statement:

```
SELECT * FROM PRODUCT_DETAILS
```

SELECT \* FROM PRODUCT\_DETAILS;

PRICE	PRODUCT_ID	NAME
100.0	1	ice-cream
150.0	2	cold-crink

(2 rows, 2 ms)

Run Run Selected Auto complete Clear SQL statement:

```
SELECT * FROM ORDER_DETAILS |
```

SELECT \* FROM ORDER\_DETAILS;

ORDER\_NO

(no rows, 1 ms)

2nd : CREATE Multiple Orders

POST

localhost:8080/api/order

Params

Authorization

Headers (8)

Body

Scripts

Se

none

form-data

x-www-form-urlencoded

raw

b

```
1  {
2      "productDetails": [
3          {
4              "productId": 1
5          },
6          {
7              "productId": 2
8          }
9      ]
10 }
11 ]
12 }
```

Body

Cookies

Headers (5)

Test Results



Pretty

Raw

Preview

Visualize

JSON



```
1  {
2      "orderNo": 1,
3      "productDetails": [
4          {
5              "productId": 1,
6              "name": "ice-cream",
7              "price": 100.0
8          },
9          {
10             "productId": 2,
11             "name": "cold-crink",
12             "price": 150.0
13         }
14     ]
15 }
```

POST

localhost:8080/api/order

Params

Authorization

Headers (8)

Body

Scripts

S

none

form-data

x-www-form-urlencoded

raw

```
1  {
2      "productDetails": [
3          {
4              "productId": 2
5          }
6      ]
7  }
```

Body

Cookies

Headers (5)

Test Results



Pretty

Raw

Preview

Visualize

JSON



```
1  {
2      "orderNo": 2,
3      "productDetails": [
4          {
5              "productId": 2,
6              "name": "cold-crink",
7              "price": 150.0
8          }
9      ]
10 }
```

Run

Run Selected

Auto complete

Clear

SQL statement:

```
SELECT * FROM ORDER_DETAILS
```

```
SELECT * FROM ORDER_DETAILS;
```

ORDER_NO
1
2

(2 rows, 1 ms)

Run

Run Selected

Auto complete

Clear

SQL statement:

```
SELECT * FROM PRODUCT_DETAILS |
```

```
SELECT * FROM PRODUCT_DETAILS;
```

PRICE	PRODUCT_ID	NAME
100.0	1	ice-cream
150.0	2	cold-crink

(2 rows, 1 ms)

### Many-to-Many: Bidirectional

- Since its Many to Many, anyone can be Owning and inverse side.

```
@Table(name = "order_details")
@Entity
public class OrderDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long orderNo;

    @ManyToMany(cascade = CascadeType.ALL)
    @JoinTable(
        name = "order_product", // new Join table name
        joinColumns = @JoinColumn(name = "order_id"), // Foreign key for Order
        inverseJoinColumns = @JoinColumn(name = "product_id") // Foreign key for Product
    )
    private List<ProductDetails> productDetails = new ArrayList<>();

    public Long getOrderNo() {
        return orderNo;
    }

    public void setOrderNo(Long orderNo) {
        this.orderNo = orderNo;
    }

    public List<ProductDetails> getProductDetails() {
        return productDetails;
    }

    public void setProductDetails(List<ProductDetails> productDetails) {
        this.productDetails = productDetails;
    }
}
```

```
@Table(name = "product_details")
@Entity
public class ProductDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long productId;

    private String name;
    private double price;

    @ManyToMany(mappedBy = "productDetails")
    @JsonIgnore
    List<OrderDetails> orders = new ArrayList<>();

    //GETTERS AND SETTERS
}
```