Spring Boot  
Basics

short line

Amit Saha

# Spring IOC Container

At the core of Spring is Spring IoC container which has 2 main features

* Loose coupling
* IOS container/ Dependency injection

We can get those using basic spring dependencies without any web dependency

<dependencies>

*<!-- https://mvnrepository.com/artifact/org.springframework/spring-core -->*

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>6.0.8</version>

</dependency>

*<!-- https://mvnrepository.com/artifact/org.springframework/spring-context -->*

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>6.0.8</version>

</dependency>

</dependencies>

## Loose coupling

Loose coupling means removing dependency between 2 classes so that they can be developed and modified independently

Let’s say we have a Traveler class which uses Car to travel

*public class* Car{

*static int* milesTravelled = 0;

*public* Car() {

milesTravelled++;

System.out.println("Car mileage " + milesTravelled);

}

*public void* move() {

System.out.println("Travel in car");

}

}

*public class* Traveler {

Car car;

*public* Traveler(Car car) {

*this*.car = car;

}

*public void* startJourney() {

*this*.car.move();

}

}

*public class* Client {

*public static void* main(String[] args) {

Car car = *new* Car();

Traveler traveler = *new* Traveler(car);

traveler.startJourney();

}

}

Now say the traveler wants to save fuel and wants to travel in Bike now the entire dependency and code of Traveler must be changed which would need a lot of code change and testing. This is **tight coupling**.

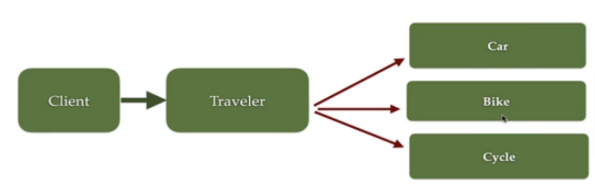
*public class* Bike {

*public void* move(){

System.out.println("Travel in bike save fuel");

}

}



To decouple the class, we create an interface vehicle and Car and Bike implement it. In the Traveler class we create a dependency on vehicles and based on the requirement we can pass a bike or car. We can change the implementation by removing the dependency on the car or the bike. This way we can pass any vehicle without changing the code.

*public interface* Vehicle {

*public void* move();

}

*public class* Car *implements* Vehicle{

*@Override*

*public void* move(){

System.out.println("Travel in car");

}

}

*public class* Bike *implements* Vehicle{

*@Override*

*public void* move(){

System.out.println("Travel in car");

}

}

*public class* Traveler {

Vehicle vehicle;

*public* Traveler(Vehicle vehicle){

*this*.vehicle = vehicle;

}

*public void* startJourney(){

*this*.vehicle.move();

}

}

*public class* Client {

*public static void* main(String[] args) {

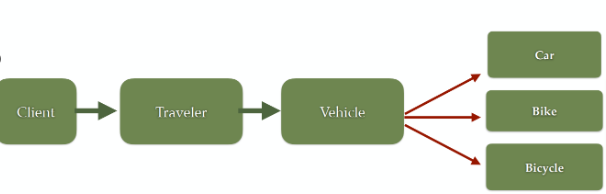
Vehicle car = *new* Car();

Traveler traveler = *new* Traveler(car);

traveler.startJourney();

}

}



## IOC container/ Dependency injection

But still in the above example we are creating objects manually with the new keyword. Car and Bike are not maintaining the state of the application These objects take up Heap memory. In real time there can be 100s or 1000s of classes which will take up space in Heap memory. Some of the classes like controllers, services etc. are the same for all requests as they are just implementation and don’t hold data. There is no point in maintaining various instances in Heap. So, it's better we let Spring create the object and manage the lifecycle of the object. This is achieved with design principal of IoC, and spring uses DI pattern to implement it

**Inversion of Control (IoC):**

* A design principle where the control of object creation and lifecycle management is transferred from the application code to an external framework (in this case, Spring)
* Instead of your code controlling object creation and dependencies, the Spring container manages these responsibilities
* Reduces tight coupling between components and improves modularity

**Dependency Injection (DI):**

* A pattern used to implement IoC in Spring
* Objects receive their dependencies from an external source (Spring container) rather than creating them internally
* Three main types of dependency injection in Spring:
  + Constructor Injection: Dependencies are provided through a class constructor
  + Setter Injection: Dependencies are set using setter methods
  + Field Injection: Dependencies are directly injected into class fields

**Spring IoC container**

1. Responsible for creating objects (Spring beans)
2. Responsible for injecting one object into another object (using DI)
3. Managing the bean's entire life-cycle - It manages the complete lifecycle of a bean from creation to destruction
4. Container uses the information provided in the configuration file (or annotations) to create the objects, and it uses dependency injections to supply the objects with their dependencies.
5. The configuration metadata is represented in XML, Java annotations, or Java code.
6. By using the IoC container, the objects are more loosely coupled and easier to test, since their dependencies can be easily swapped out with mock objects.

Spring achieves Dependency Injection (DI) and Inversion of Control (IoC) through its core container, primarily using these key mechanisms:

* IoC Container Types:
  + **BeanFactory container:** Basic container responsible for creating and configuring spring beans (lazy loading) and maintaining its life cycle
  + **ApplicationContext container:** Advanced container with additional enterprise-specific features
    - Responsible for creating the beans, configuring the beans
    - and managing the bean's entire life cycle.
    - Enterprise Application Features:
      * Internationalization support
      * Event publication mechanism
      * Flexible dependency injection
      * Aspect-Oriented Programming (AOP) integration
      * Comprehensive bean lifecycle management
      * Multiple bean scoping options
* Dependency Injection Methods:
  + Constructor Injection
  + Setter Injection
  + Field Injection

SpringBoot uses ApplicationContext containers

## Creating Spring Bean

Spring beans can be created using 2 ways

1. Java Based Configuration
2. Annotation based configuration

### Java Based configuration

Steps for Java based configuration

1. Create Configuration class with @Configuration annotation
2. Create method and annotated it with @Bean annotation
3. Create Spring IoC Container (ApplicationContext) and Retrieve Spring bean from Spring IoC container.

In the App config class, we add @Configuration and @Bean annotation, for object initialization. Spring container also takes care of dependency of Traveler on Vehicle by injecting Car in Traveler class. Since there are 2 Traveler beans, one with car another with Bike we add property name in bean to distinguish them

*@Configuration*

*public class AppConfiguration {*

*@Bean*

*public Vehicle car(){*

*return new Car();*

*}*

*@Bean*

*public Vehicle bike(){*

*return new Bike();*

*}*

*@Bean(name = "travelerWithCar")*

*public Traveler travelerWithCar(){*

*return new Traveler(car());*

*}*

*@Bean(name = "travelerWithBike")*

*public Traveler travelerWithBike(){*

*return new Traveler(bike());*

*}*

*}*

In the car class and constructor, we add a milesTravelled static variable to check how many times the car constructor is called.

*public class* Car *implements* Vehicle {  
  
  
 *static int* carInstanceCount = 0;  
  
 *public* Car() {  
 carInstanceCount++;  
 System.out.println("Car instance count " + carInstanceCount);  
 }  
  
 *@Override  
 public void* move() {  
 System.out.println("Travel in car");  
 }  
}

Now in Client class we create Car class twice using Spring and once with normal new. Even though we call Car twice with spring, the constructor is called only the 1st time. Then it is managed by Spring container

Since there are 2 traveler beans we have to pass the bean name for Traveler bean.

*public class* JavaBasedSpringBeanClient {  
 *public static void* main(String[] args) {  
 ApplicationContext applicationContext = *new* AnnotationConfigApplicationContext(AppConfiguration.*class*);  
  
 Car car1 = applicationContext.getBean(Car.*class*);  
 Car car2 = applicationContext.getBean(Car.*class*); *// will not trigger constructor again* Bike bike1 = applicationContext.getBean(Bike.*class*);  
 Bike bike2 = applicationContext.getBean(Bike.*class*); *// will not trigger constructor again* Vehicle car0 = *new* Car(); *// Will trigger constructor again  
 new* Traveler(car0).startJourney();  
  
 Traveler traveler = (Traveler) applicationContext.getBean("travelerWithCar");  
 traveler.startJourney();  
 Traveler traveler1 = (Traveler) applicationContext.getBean("travelerWithCar");  
 traveler1.startJourney();  
 Traveler traveler2 = (Traveler) applicationContext.getBean("travelerWithBike");  
 traveler2.startJourney();  
  
 }  
}

### Annotation Based Configuration

How to create Annotation based configuration

1. @ComponentScan : This will mark the package eligible for scan by Spring container. In those packages all classes annotated with @Component will register as Spring bean.
2. @Component : For the classes for which bean needs to be created add @Component annotation. This will register the class as Spring bean
3. @Autowired : Inject the Springbean
4. @Qualifier : To distinguish between different Beans of same class
5. Create Spring IoC container (ApplicationContext) and retrieve Spring bean from Spring Ioc container.

Example We first define Vehicle interface

*public interface* Vehicle {

*public void* move();

}

In the configuration class we remove the beans and add @ComponenetScan with the base package

*@Configuration*

*@ComponentScan*(basePackages = "com.saha.amit.d\_annotationBasedConfiguration")

*public class* AppConfiguration {

}

Then we add @Component in various beans and provide a name for the bean. When you declare a bean without specifying a name for it, the Spring container uses a default naming convention to generate a name for that bean. **Default Naming Convention:** The default bean name is the unqualified class name, with the **first letter changed to lowercase**.

*@Component*("car")  
*@Primary  
public class* Car *implements* Vehicle {  
 *static int* instanceCount = 0;  
  
 *public* Car() {  
 instanceCount++;  
 System.out.println("Car instance count " + instanceCount);  
 }  
  
 *@Override  
 public void* move() {  
 System.out.println("Travel in car");  
 }  
}

*@Component*("bike")

*public class* Bike *implements* Vehicle {

*@Override*

*public void* move(){

System.out.println("Travel in Bike");

}

}

Since there are multiple vehicle interfaces, spring will not know which bean to pass. So, we can mark one of the bean with @Primary, or we can pass which bean to load by passing @Qualifer annotation and mention which bean we want to load. This will overwrite @Primary

*@Component*("traveler")

*public class* Traveler {

Vehicle vehicle;

*@Autowired*

*public* Traveler(*@Qualifier*("car") Vehicle vehicle){

*this*.vehicle = vehicle;

}

*public void* startJourney(){

*this*.vehicle.move();

}

}

With @Primary or @Qualifier we are tied with our implementation during compile time. If we want to make it dynamic we can use factory implementation. We can create a VehicleFactory with a getVehicle method which will return the bean based on String passed

*@Component  
public class* VehicleFactory {  
 *@Autowired  
 private* ApplicationContext context;  
  
 *public* Vehicle getVehicle(String type) {  
 *return* context.getBean(type, Vehicle.*class*);  
 }  
}

And use the factory class in

*public class* AnnotationBasedSpringClient {  
 *public static void* main(String[] args) {  
 ApplicationContext applicationContext = *new* AnnotationConfigApplicationContext(AppConfiguration.*class*);  
  
  
 Car car1 = applicationContext.getBean(Car.*class*);  
 Car car2 = applicationContext.getBean(Car.*class*); *// will not trigger constructor again* Traveler traveler = applicationContext.getBean(Traveler.*class*);  
 traveler.startJourney();  
  
 Traveler traveler2 = applicationContext.getBean(Traveler.*class*);  
 traveler2.startJourney();  
  
 Traveler traveler3 = applicationContext.getBean(Traveler.*class*);  
 traveler3.setVehicle("car"); *// Creating bean dynamically using factory method* traveler3.startJourney();  
 }  
}

### Stereotype Annotations

Annotation based configuration provides some inbuilt Annotation like below

Stereotype annotations

1. These annotations are used to create Spring beans automatically in the application context (Spring IoC container)
2. The main stereotype annotation is @Component.
3. By using this annotation, Spring provides more Stereotype meta annotations such as @Service, @Repository and @Controller
4. @Service annotation is used to create Spring beans at the Service layer
5. @Repository is used to create Spring beans for the repositoriesat the DAO layer
6. @Controller is used to create Spring beans at the controller layer



*@Controller*

*public class* DemoController {

*private* DemoService demoService;

*public* String hello(){

*return* "hello controller";

}

}

*@Service*

*public class* DemoService {

*public* String hello(){

*return* "Hello service";

}

}

*@Repository*

*public class* DemoRepository {

*public* String hello(){

*return* "Hello repository";

}

}

*@Configuration*

*@ComponentScan*(basePackages = "com.spring.core")

*public class* AppConfig {

}

*public class* DemoClient {

*public static void* main(String[] args) {

ApplicationContext applicationContext = *new* AnnotationConfigApplicationContext(AppConfig.*class*);

DemoController demoController = applicationContext.getBean(DemoController.*class*);

System.out.println(demoController.hello());

DemoService demoService = applicationContext.getBean(DemoService.*class*);

System.out.println(demoService.hello());

DemoRepository demoRepository = applicationContext.getBean(DemoRepository.*class*);

System.out.println(demoRepository.hello());

}

}

The default scope of Spring beans created by stereotype annotations (such as @Component, @Service, @Repository, and @Controller) is **singleton**.

### Dependency Injection types

This is of 3 type



#### Constructor based DI

This is what we have been doing normally. Defining the beans and setting dependency using @Autowired annotation setting the dependency in. When there is only one constructor @Autowired is optional as the spring constructor injects it.

*@Component*

*public class* EmailService *implements* MessageService{

*@Override*

*public void* sendMessage(String message){

System.out.println(message);

}

}

*@Component*("smsService")

*public class* SMSService *implements* MessageService{

*@Override*

*public void* sendMessage(String message){

System.out.println(message);

}

}

*@Component*

*public class* MessageSender {

*private* MessageService messageService;

*@Autowired // Optional when we have only one constructor*

*public* MessageSender(*@Qualifier*("emailService") MessageService messageService){

*this*.messageService= messageService;

}

*public void* sendMessage(String message){

*this*.messageService.sendMessage(message);

}

}

#### Setter based DI

*@Component*

*public class* MessageSender {

*private* MessageService messageService;

*//Setter based DI*

*@Autowired*

*public void* setMessageService(*@Qualifier*("emailService") MessageService messageService) {

*this*.messageService = messageService;

}

*public void* sendMessage(String message){

*this*.messageService.sendMessage(message);

}

}

#### Field based DI

In case of field injection, spring container uses reflection to inject the

*@Component*

*public class* MessageSender {

*@Autowired*

*@Qualifier("emailService")*

*private* MessageService messageService;

*public void* sendMessage(String message){

*this*.messageService.sendMessage(message);

}

}

#### Which one to use

Field injection drawbacks

1. Can’t make the field immutable by making it final. It will throw compilation errors. Can be done only by Constructor injection.
2. Spring container uses a reflection to inject the dependencies, which is more costly than constructor based and setter based injection.
3. Very tightly coupled with Spring IoC container, which makes unit testing difficult by mockling values.

These leaves us with 2 option Constructor based DI and Setter based DI

When we have mandatory dependency it's better to use Constructor injection as it ensures dependencies are present during object creation. If the dependencies are not mandatory, better to use setter injection. As the object will be created first then the dependency

*class* DependencyA{}

*class* DependencyB{}

*class* DependencyC{}

*public class* Demo {

*private* DependencyA dependencyA;

*private* DependencyB dependencyB;

*private* DependencyC dependencyC;

*public* Demo(DependencyA dependencyA, DependencyB dependencyB, DependencyC dependencyC) {

*this*.dependencyA = dependencyA;

*this*.dependencyB = dependencyB;

*this*.dependencyC = dependencyC;

}

*public void* setDependencyA(DependencyA dependencyA) {

*this*.dependencyA = dependencyA;

}

*public void* setDependencyB(DependencyB dependencyB) {

*this*.dependencyB = dependencyB;

}

*public void* setDependencyC(DependencyC dependencyC) {

*this*.dependencyC = dependencyC;

}

}

In case of cyclic dependency where A depends on B and B depends on A it's better to used setter injection

*class* DependencyA{

*private* DependencyB dependencyB;

*public* DependencyA(DependencyB dependencyB) {

*this*.dependencyB = dependencyB;

}

*public void* setDependencyB(DependencyB dependencyB) {

*this*.dependencyB = dependencyB;

}

}

*class* DependencyB{

*private* DependencyA dependencyA;

*public* DependencyB(DependencyA dependencyA) {

*this*.dependencyA = dependencyA;

}

*public void* setDependencyA(DependencyA dependencyA) {

*this*.dependencyA = dependencyA;

}

}

We can also make dependency immutable by making it final in constructor injection. We can’t do Setter injection by marking the dependency as final.

Also we don’t need to do null check as dependencies are initialized when class is initialized.

*private final* DependencyA dependencyA;

*private final* DependencyB dependencyB;

*private final* DependencyC dependencyC;

*public* Demo(DependencyA dependencyA, DependencyB dependencyB, DependencyC dependencyC) {

*this*.dependencyA = dependencyA;

*this*.dependencyB = dependencyB;

*this*.dependencyC = dependencyC;

}

## Bean Life Cycle

The Spring Bean lifecycle is the sequence of events or actions that occur from the creation of a bean instance to its destruction in a Spring application context. Let's break down the lifecycle and the related actions demonstrated in your example:

**Spring Bean Lifecycle Overview**

1. Instantiation: Spring creates a new instance of the bean by calling its constructor.
2. Dependency Injection: Dependencies of the bean are injected via constructor, setters, or field injection.
3. Bean Initialization: After dependencies are injected, any custom initialization logic (if defined) is executed. This may include:
   1. @PostConstruct annotated methods. Present in jakarta.annotation dependency not spring
   2. Custom initialization methods defined with the initMethod attribute in @Bean annotation or the InitializingBean interface.
4. Bean Usage: The bean is now ready for use within the application.
5. Bean Destruction: Before the application context is closed, the bean's custom destruction logic (if defined) is executed. This may include:
   1. @PreDestroy annotated methods. Present in jakarta.annotation dependency not spring
   2. Custom destroy methods defined with the destroyMethod attribute in @Bean annotation or the DisposableBean interface.

*public class* Address {  
 *public void* print() {  
 System.out.println("Address class method called ...");  
 }  
}

*public class* Student {  
 *private* Address address;  
  
 *public* Student(Address address){  
 *this*.address = address;  
 }  
  
 *public void* print(){  
 System.out.println("Student class method called ...");  
 address.print();  
 }  
  
 *public void* init(){  
 System.out.println("Initialization logic");  
 }  
  
 *public void* destroy(){  
 System.out.println("Destruction logic");  
 }  
  
}

*@Configuration  
public class* AppConfig{  
  
 *@Bean*(name = "addressBean")  
 *public* Address address(){  
 *return new* Address();  
 }  
  
 *@Bean*(name = {"studentBean","studentDemo"}, initMethod = "init", destroyMethod = "destroy")  
 *public* Student student(){  
 *return new* Student(address());  
 }  
}

*public class* BeanAnnotationDemo {  
  
 *public static void* main(String[] args) {  
 *try*(*var* applicationContext = *new* AnnotationConfigApplicationContext(AppConfig.*class*)){  
  
 */\*List all the bean names  
 String[] beanNames = applicationContext.getBeanDefinitionNames();  
 for (String bean: beanNames){  
 System.out.println("Beans " +bean);  
 }\*/* Student student = (Student) applicationContext.getBean("studentBean");  
 student.print();  
 }  
 }  
}

## Bean Scope

The latest version of the Spring framework defines 6 types of scopes

* singleton
* prototype
* request
* session
* application
* websocket

The last four scopes mentioned, request, session, application and websocket, are only available in a web-aware application.

* @Scope annotation is used to define a scope of the bean
* We use @Scope to define the scope of a @Component class or a @Bean definition.
* Singleton: only one instance of the bean is created and shared across the entire application. This is the default scope.
* Prototype: a new instance of the bean is created every time it is requested.

*@Component  
@Scope*(value = ConfigurableBeanFactory.SCOPE\_PROTOTYPE)  
*public class* PrototypeBean {  
 *public* PrototypeBean() {  
 System.out.println("PrototypeBean constructor ..");  
 }  
}

*@Component  
@Scope*(value = ConfigurableBeanFactory.SCOPE\_SINGLETON)  
*public class* SingletonBean {  
 *public* SingletonBean() {  
 System.out.println("SingletonBean constructor ..");  
 }  
}

*@Configuration  
@ComponentScan*(basePackages = "com.saha.amit.h\_beanScope")  
*public class* AppConfiguration {  
}

*public class* BeanScopeClient {  
 *public static void* main(String[] args) {  
 *var* applicationContext = *new* AnnotationConfigApplicationContext(AppConfiguration.*class*);  
 PrototypeBean prototypeBean = applicationContext.getBean(PrototypeBean.*class*);  
 PrototypeBean prototypeBean1 = applicationContext.getBean(PrototypeBean.*class*); *//Will call constructor* SingletonBean singletonBean = applicationContext.getBean(SingletonBean.*class*);  
 SingletonBean singletonBean1 = applicationContext.getBean(SingletonBean.*class*); *//Will not call constructor* }  
  
}

## Spring @Lazy Annotation

By default, Spring creates all singleton beans eagerly at the startup/bootstrapping of the application context. The @Lazy annotation in Spring is used to delay the initialization of a bean until it is actually needed, rather than at application startup. By default, Spring initializes the singleton beans eagerly at startup, but using @Lazy, you can change this behavior.

* You can load the Spring beans lazily (on-demand) using @Lazy annotation
* @Lazy annotation can used with @Configuration, @Component and @Bean annotations

Use Cases of @Lazy

* To improve application startup performance by deferring the creation of beans that are not immediately needed.
* To avoid unnecessary resource allocation for beans that may not be used.

Eager initialization is recommended: to avoid and detect all possible errors immediately rather than at runtime.

*@Configuration  
@ComponentScan*(basePackages = "com.saha.amit.i\_lazyBeanLoad")  
*public class* AppConfiguration {  
}

*@Component  
public class* EagerLoader {  
 *public* EagerLoader() {  
 System.out.println("EagerLoader..");  
 }  
}

*@Component  
@Lazy  
public class* LazyLoader {  
 *public* LazyLoader() {  
 System.out.println("LazyLoader ..");  
 }  
}

*public class* Client {  
 *public static void* main(String[] args) {  
 System.out.println("Starting application context...");  
 *var* applicationContext = *new* AnnotationConfigApplicationContext(AppConfiguration.*class*); *//Eager will be loaded now* System.out.println("Application context started!");  
  
 EagerLoader eagerLoader = applicationContext.getBean(EagerLoader.*class*);  
 LazyLoader lazyLoader = applicationContext.getBean(LazyLoader.*class*); *//Lazy will be loaded now* }  
}

# Spring Boot Basics

**Key Features of Spring Boot**

1. **Auto-Configuration**

* Spring Boot automatically configures the application based on the dependencies added in your pom.xml (Maven) or build.gradle (Gradle).
* Example: If spring-boot-starter-web is included, it configures a web application with an embedded Tomcat server.

1. **Starter Dependencies**

* Spring Boot provides **starter dependencies** to simplify dependency management.
* Example:
  + spring-boot-starter-web: For building web applications (includes Spring MVC, Tomcat).
  + spring-boot-starter-data-jpa: For JPA-based database access.
  + spring-boot-starter-security: For security features.

1. **Embedded Servers**

* Spring Boot comes with embedded servers like **Tomcat**, **Jetty**, and **Undertow** to run web applications without external server configuration.
* Example: Run your application by executing the main method — no need to deploy WAR files to an external server.

1. **Spring Boot Actuator**

* Provides production-ready features like monitoring and management endpoints.
* Example:
  + /actuator/health: Provides application health status.
  + /actuator/metrics: Displays application metrics.

1. **Spring Boot CLI**

* A command-line tool to quickly prototype applications using **Groovy** scripts.
* Example: spring run app.groovy

1. **Externalized Configuration**

* Spring Boot supports flexible configuration through **application.properties** or **application.yml**.
* Configuration can also be overridden with environment variables, command-line arguments, or .env files.

Example (application.properties):

server.port=8081

spring.datasource.url=jdbc:mysql://localhost:3306/mydb

spring.datasource.username=root

spring.datasource.password=pass

1. **Spring Initializer**

* A web-based tool or CLI to bootstrap Spring Boot applications with necessary dependencies.
* Available at [start.spring.io](https://start.spring.io).

1. **Profiles and Environment Management**

* Use **profiles** to configure different environments like development, staging, and production.

Example:

application-dev.properties for development.

application-prod.properties for production.

Activate a profile:

spring.profiles.active=dev

1. **Developer Tools (spring-boot-devtools)**

Provides features like hot reloading, automatic restart, and live reload during development

1. **Opinionated Defaults**

* Spring Boot provides sensible defaults for most configurations, which can be overridden if needed.
* Example: Default server port is 8080, default error pages are provided, etc.

1. **Easy Testing**
   * Spring Boot simplifies testing with annotations like:
     + @SpringBootTest: For integration testing.
     + @WebMvcTest: For testing Spring MVC components.
     + @DataJpaTest: For testing JPA repositories.
2. **Built-in Security** With spring-boot-starter-security, Spring Boot automatically secures all endpoints with basic authentication
3. **Easy Integration with Databases**

* Provides support for popular databases and ORMs like:
  + **Spring Data JPA** for relational databases.
  + **Spring Data MongoDB** for MongoDB.
  + Built-in H2 database for testing

**Advantages of Using Spring Boot**

1. **Reduces Boilerplate Code**: Less XML/Java configuration.
2. **Faster Development**: Auto-configuration and starter templates make setup easier.
3. **Microservices-Friendly**: Simplifies microservice development with features like embedded servers and actuator.
4. **Production-Ready**: Actuator provides health checks, metrics, and monitoring.
5. **Extensive Community Support**: Large ecosystem and active community.

## POM.xml

Starter project

<parent>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-parent</artifactId>  
 <version>3.2.2</version>  
 <relativePath/> *<!-- lookup parent from repository -->*</parent>

<groupId>com.saha.amit</groupId>  
<artifactId>springBootBasic</artifactId>  
<version>${project.version}</version>  
<packaging>jar</packaging>  
<name>springBoot-Basics</name>  
<description>Basic introduction to Spring boot</description>  
  
<properties>  
 <maven.compiler.source>21</maven.compiler.source>  
 <maven.compiler.target>21</maven.compiler.target>  
 <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  
 <project.version>999</project.version>  
</properties>

<build>  
 <plugins>  
 <plugin>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-maven-plugin</artifactId>  
 <configuration>  
 <mainClass>com.saha.amit.SpringBootBasicMainApplication</mainClass>  
 <layout>JAR</layout>  
 <image>  
 <name>justamitsaha/${project.artifactId}:kube\_3</name>  
 </image>  
 <excludes>  
 <exclude>  
 <groupId>org.projectlombok</groupId>  
 <artifactId>lombok</artifactId>  
 </exclude>  
 </excludes>  
 </configuration>  
 </plugin>  
 </plugins>  
</build>

mvn clean

## Spring Boot starter projects

Spring Boot starter projects are convenient dependency descriptors. To build a specific kind of application, you might need a set of dependencies, for example, to build. a web application. You need a set of dependencies. All of these are predefined in startup projects. Spring Boot provides you with a variety of startup projects. Like Spring web, Spring JPA, Spring web flux

For e.g. Spring web provides

1. Web project
2. Spring web MVC
3. Tom cat
4. JSON POJO conversion
5. Testing framework.

**1. API Documentation**

OpenAPI generates detailed and interactive API documentation automatically. Using tools like Swagger UI or ReDoc, you can visualize the API endpoints, their descriptions, request/response details, and more.

* **Example:** Interactive documentation allows developers to test API endpoints directly in the browser, reducing the need for external tools like Postman.

**2. Standardized API Description**

OpenAPI uses a standardized YAML or JSON file (commonly called an OpenAPI Specification or OAS) to describe API operations, including:

* **Endpoints** (/users, /products/{id}).
* **HTTP methods** (GET, POST, PUT, DELETE).
* **Request/response bodies**, parameters, headers, and status codes.
* **Authentication** mechanisms (e.g., OAuth 2.0, API keys, Bearer tokens).

**3. API Versioning**

OpenAPI supports versioning of APIs to help manage breaking changes and backward compatibility. You can add version details to the path (e.g., /v1/users) or metadata.

**4. Code Generation**

The OpenAPI Specification can generate client SDKs, server stubs, and API documentation for many programming languages (e.g., Java, Python, JavaScript) using tools like:

* **Swagger Codegen**
* **OpenAPI Generator**

This speeds up development by automating boilerplate code.

**5. Schema Validation**

OpenAPI uses JSON Schema to validate request/response bodies. This ensures that the data passed to and from your API conforms to a defined structure.

* **Example:** Ensuring that an incoming payload contains mandatory fields like id, name, etc.

**6. Rich Metadata Support**

You can define comprehensive metadata about the API, including:

* API title and description.
* Contact information for support.
* Licensing details (e.g., MIT, Apache 2.0).
* Terms of service links.

**7. Support for API Security**

OpenAPI allows the specification of various authentication methods:

* **HTTP Authentication** (e.g., Basic, Bearer).
* **API Keys** (passed via headers, query parameters, or cookies).
* **OAuth 2.0** with flows like Implicit, Password, Client Credentials, and Authorization Code.

**8. Response Documentation**

Each endpoint can have detailed documentation about possible HTTP status codes and responses. For example:

* **200 OK**: The request was successful, with a defined response body schema.
* **404 Not Found**: Resource not found, with a description of the error.

**9. Parameterization**

OpenAPI allows parameters to be defined at different levels:

* **Path Parameters**: Variables in the endpoint URL (e.g., /users/{id}).
* **Query Parameters**: Filters or options (e.g., GET /users?age=25).
* **Headers**: Custom headers for additional metadata.
* **Cookie Parameters**: API state management.

**10. Examples and Default Values**

You can provide example values for request bodies, parameters, and responses. This helps developers understand the expected input/output quickly.

## Open API specification

OpenAPI, formerly known as Swagger, is a specification for describing, documenting, and defining RESTful APIs in a machine-readable format. It provides a standard way to share API design, enabling consistency and ease of use for developers. OpenAPI features significantly streamline the process of API development, testing, and documentation. Here’s a breakdown of its key features:

## Spring Auto Configuration

Spring Boot's auto-configuration is a feature that automatically configures your application based on the dependencies present on the classpath.

* It eliminates the need for manual configuration and boilerplate code, allowing developers to focus on writing business logic.
* Auto-configuration works by scanning the classpath for specific libraries (like spring-web, spring-data-jpa) and providing sensible default configurations based on those libraries.

The automatic configuration is generated based on which frameworks are in the class path. In the pom XML, we can add a lot of startup projects, and these bring in a lot of frameworks, depending on the frameworks which are present in the class path. All the auto configuration logic is defined in a specific jar which you would be able to see in your dependencies in a jar boot.autoconfigure.something.jar. Different frameworks will have different autoconfigure.jar. If we look in those, we will find auto configuration



In order to view more details on auto configuration go to application.properties and change logging level to

logging.level.org.springframework= debug,

or

debug=true

The default logging level is info and info prints very, very little information. However, when we changed it to debug, you would see that there is a lot of information which is printed in. If you look at the logs in depth, you will see that there is something called **conditions Evaluation** report. There are positive matches and there are negative matches. Negative matches are things which did not get auto configured. Positive matches are the things which were auto configured for you. We can search with “Positive” and “Negative” in console to find these.





From the console we can see the auto configuration classes

*@AutoConfigureOrder*(Integer.MIN\_VALUE)

*@AutoConfiguration*(

after = {ServletWebServerFactoryAutoConfiguration.*class*}

)

*@ConditionalOnWebApplication*( type = Type.SERVLET)

*@ConditionalOnClass*({DispatcherServlet.*class*})

*public class* DispatcherServletAutoConfiguration {

*public static final* String DEFAULT\_DISPATCHER\_SERVLET\_BEAN\_NAME = "dispatcherServlet";

*public static final* String DEFAULT\_DISPATCHER\_SERVLET\_REGISTRATION\_BEAN\_NAME = "dispatcherServletRegistration";

We can see the condition when this auto configuration is enabled. It is done when the application is a type of web application and the dispatcher servlet is present in classpath

*@ConditionalOnWebApplication*( type = Type.SERVLET)

*@ConditionalOnClass*({DispatcherServlet.*class*})

## Spring Boot configuration

Spring Boot lets you externalize your configuration so that you can work with the same application code in different environments. You can use a variety of external configuration sources, including Java properties files, YAML files, environment variables, and command-line arguments. Properties are considered in the following order (with values from lower items overriding earlier ones)

1. Properties present inside files like application.properties
2. OS Environmental variables
3. Java System properties (System.getProperties())
4. JNDI attributes from java:comp/env
5. ServletContext init parameters
6. ServletConfig init parameters
7. Command line arguments

Java values can be set using @Value annotations

*@Value*("${payment.salary}")  
*private int* salary;  
  
*@Value*("${payment.message}")  
*private* String message;

Java Objects can be set using

*@ConfigurationProperties*(prefix = "user")  
*public class* UserProfile {  
 *private* String names;  
 *private* String ssn;  
 *private int* age;  
 *private* List<String> address;  
}

These values can be defined in application.properties

user.names=John Doe  
user.ssn=653156989  
user.age= 40  
user.address = 9674 Spencer Street, Apt 111, Torrance, California, 90503

payment.salary=4200  
payment.message= $ is the salary

We can define multiple profiles in properties file using spring profiles. Say we want to maintain different profiles for different environments we can do that with maintaining different profiles files application-dev.properties, application-prod.properties

application-prod.properties

user.names=John Doe  
user.ssn=653156989  
user.age= 40  
user.address = 9674 Spencer Street, Apt 111, Torrance, California, 90503  
payment.salary=4200  
payment.message= $ is the salary

application-dev.properties

user.names=Jane Doe  
user.ssn=85657878  
user.age= 38  
user.address = 3654 Park-wood Boulevard, Apt 1313, Frisco, Texas, 75020  
payment.salary=4300  
payment.message= $ is the salary

and in application.properties we can refer which profile we want to load. And Java variables will load values from properties file

spring.profiles.active=dev  
management.endpoints.web.exposure.include=\*

Same thing we can achieve is application.yml in file as yml supports multiple files in single file

spring:  
 profiles:  
 active: dev *# Set the active profile to 'dev' or 'prod' here*management:  
 endpoints:  
 web:  
 exposure:  
 include: "\*"  
  
*# Profile-specific configurations*---  
spring:  
 config:  
 activate:  
 on-profile: dev  
user:  
 names: John Doe  
 ssn: 653156989  
 age: 40  
 address: 9674 Spencer Street, Apt 111, Torrance, California, 90503  
payment:  
 salary: 4200  
 message: " $ is the salary"  
  
---  
spring:  
 config:  
 activate:  
 on-profile: prod  
user:  
 names: Jane Doe  
 ssn: 85657878  
 age: 38  
 address: 3654 Park-wood Boulevard, Apt 1313, Frisco, Texas, 75020  
payment:  
 salary: 4300  
 message: " $ is the salary"

Command line arguments these will overwrite properties file

How to set environment variables

export myCustomProp=98.80.216.66 ##unix  
set myCustomProp =98.80.216.66 ## windows

*@Autowired  
private* Environment environment;  
  
  
*@GetMapping*("env")  
*public* String getEnv() {  
 log.info(environment.getProperty("JAVA\_HOME") + environment.getProperty("myCustomProp"));  
 *return* environment.getProperty("JAVA\_HOME") + environment.getProperty("myCustomProp");  
}

Set using CLI

mvn clean

mvn clean install

java -jar .\springBootBasic-999.jar --spring.profiles.active=prod --myCustomProp=hello

java -DmyCustomProp=hello -jar .\springBootBasic-999.jar --spring.profiles.active=prod

## Spring boot actuator

Add dependency

<dependency>

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

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

All the list of URLs exposed are displayed with URL <http://localhost:8080/actuator>

By default only <http://localhost:8080/actuator/health> is exposed. For exposing other URI add below in properties

Some important URI exposed

<http://localhost:8080/actuator/beans> - list of beans

<http://localhost:8080/actuator/configprops> list of properties available to override

<http://localhost:8080/actuator/env> environment details

<http://localhost:8080/actuator/metrics> then add the parameters returned in URI to get granular details for that parameter like <http://localhost:8080/actuator/metrics/http.server.requests>

# Spring DATA

Spring Data’s mission is to provide a familiar and consistent, Spring-based programming model for data access while still retaining the special traits of the underlying data store. It makes it easy to use data access technologies, relational and non-relational databases, map-reduce frameworks, and cloud-based data services. This is an umbrella project which contains many subprojects that are specific to a given database

## Spring JDBC template

Spring JdbcTemplate provides multiple methods to execute queries, each suited to different scenarios. Below is a comprehensive list of the most common ways:

### 1. Execute a Query to Fetch Data

#### **a. queryForObject**

Fetch a single row as a single value or an object.

* **Usage**: When the query returns exactly one row.

**Example**:  
java  
Copy code  
String sql = "SELECT COUNT(\*) FROM Product";

Integer count = jdbcTemplate.queryForObject(sql, Integer.class);

With a custom object:  
java  
Copy code  
String sql = "SELECT name FROM Product WHERE product\_uuid = ?";

String productName = jdbcTemplate.queryForObject(sql, new Object[]{1L}, String.class);

#### **b. query**

Fetch multiple rows and map them to objects.

* **Usage**: When the query returns multiple rows.

**Example**:  
java  
Copy code  
String sql = "SELECT \* FROM Product";

List<Product> products = jdbcTemplate.query(sql, new ProductRowMapper());

Using a Lambda:  
java  
Copy code  
List<Product> products = jdbcTemplate.query(sql, (rs, rowNum) -> {

Product product = new Product();

product.setId(rs.getLong("product\_uuid"));

product.setName(rs.getString("name"));

product.setPrice(rs.getDouble("price"));

return product;

});

#### **c. queryForList**

Fetch multiple rows as a List of maps or simple values.

**Example**:  
java  
Copy code  
String sql = "SELECT name FROM Product";

List<String> productNames = jdbcTemplate.queryForList(sql, String.class);

#### **d. queryForMap**

Fetch a single row as a key-value map.

**Example**:  
java  
Copy code  
String sql = "SELECT \* FROM Product WHERE product\_uuid = ?";

Map<String, Object> product = jdbcTemplate.queryForMap(sql, 1L);

### 2. Execute an Update

#### **a. update**

Perform INSERT, UPDATE, or DELETE operations.

* **Usage**: For single DML statements.

**Example**:  
java  
Copy code  
String sql = "UPDATE Product SET price = ? WHERE product\_uuid = ?";

int rowsAffected = jdbcTemplate.update(sql, 199.99, 1L);

#### **b. batchUpdate**

Perform batch operations for efficiency.

* **Usage**: When inserting/updating multiple rows in a single operation.

**Example**:  
java  
Copy code  
String sql = "INSERT INTO Product (name, price) VALUES (?, ?)";

jdbcTemplate.batchUpdate(sql, new BatchPreparedStatementSetter() {

@Override

public void setValues(PreparedStatement ps, int i) throws SQLException {

ps.setString(1, products.get(i).getName());

ps.setDouble(2, products.get(i).getPrice());

}

@Override

public int getBatchSize() {

return products.size();

}

});

### 3. Execute Arbitrary SQL Statements

#### **a. execute**

Execute a custom SQL statement that doesn’t return a result set (e.g., DDL).

**Example**:  
java  
Copy code  
String sql = "CREATE TABLE Temp (id INT PRIMARY KEY, name VARCHAR(255))";

jdbcTemplate.execute(sql);

#### **b. Callable for Stored Procedures**

**Example**:  
java  
Copy code  
jdbcTemplate.execute(connection -> {

CallableStatement callableStatement = connection.prepareCall("{call some\_procedure(?)}");

callableStatement.setString(1, "param");

return callableStatement;

});

### 4. Key Generation

#### **a. Using GeneratedKeyHolder**

Retrieve generated keys (e.g., primary keys).

**Example**:  
java  
Copy code  
String sql = "INSERT INTO Product (name, price) VALUES (?, ?)";

KeyHolder keyHolder = new GeneratedKeyHolder();

jdbcTemplate.update(connection -> {

PreparedStatement ps = connection.prepareStatement(sql, Statement.RETURN\_GENERATED\_KEYS);

ps.setString(1, "New Product");

ps.setDouble(2, 99.99);

return ps;

}, keyHolder);

Long generatedId = keyHolder.getKey().longValue();

### 5. Named Parameters

Use NamedParameterJdbcTemplate for better readability when working with queries with multiple parameters.

**Example**:  
java  
Copy code  
NamedParameterJdbcTemplate namedJdbcTemplate = new NamedParameterJdbcTemplate(jdbcTemplate);

String sql = "SELECT \* FROM Product WHERE price > :price";

Map<String, Object> params = Map.of("price", 100.0);

List<Product> products = namedJdbcTemplate.query(sql, params, new ProductRowMapper());

### 6. Custom Row Mapping

#### **a. Using RowMapper**

Custom row mapping for complex objects.

**Example**:  
java  
Copy code  
public class ProductRowMapper implements RowMapper<Product> {

@Override

public Product mapRow(ResultSet rs, int rowNum) throws SQLException {

Product product = new Product();

product.setId(rs.getLong("product\_uuid"));

product.setName(rs.getString("name"));

product.setPrice(rs.getDouble("price"));

return product;

}

}

### Choosing the Right Method

* **Simple Queries**: Use queryForObject, queryForList, or queryForMap.
* **Batch Operations**: Use batchUpdate for better performance.
* **Dynamic SQL**: Consider NamedParameterJdbcTemplate for improved readability.
* **Complex Object Mapping**: Use custom RowMapper with query.

## JPA

Java/Jakarta Persistence API is a specification for object-relational mapping in Java. It provides specifications to map objects to Database rows. So that we can work with Java objects instead of SQL queries. In JPA a single object can represent a row in a database table or multiple tables.

ORM stands for Object-Relational Mapping, a programming technique that converts data between a relational database and an object-oriented programming language

Hibernate is the default JPA provider for Spring Boot. It is a java based ORM tool that provides a framework for mapping application domain objects to DB tables and vice versa

Spring data JPA is an abstraction layer on top of JPA to reduce the amount of boilerplate code required to implement the DAO layer. JPA also provides different ways of querying the data from the database. By default with JPA, You don't need to write any queries at all. But in certain situations you might want to write a few queries and in those situations there are three different options that are present. One is called JPQL. In JPQL, we write queries using entities. So in JPQL, we write queries using the Java entities. The other option is to not use SQL at all. You can use something called criteria query to build the entire query using Java API. This is called criteria queries.The last option JPA provides, is to use your native queries directly is to use your SQL directly



Generation types

1. GenerationType.**AUTO** –>This is the default strategy if you don't explicitly specify a generation type. The JPA provider chooses the most suitable strategy based on the underlying database.
2. GenerationType.**IDENTITY** –> Relies on the database to generate unique values, typically using an auto-incrementing column
3. GenerationType.**SEQUENCE** –> Uses a database sequence to generate unique values. Requires specifying a sequence name and optionally an initial value and increment. Slightly slower compared to IDENTITY due to the extra database round trip But it's good when we do insert in bulk. Offers more control over the sequence generation process, including:

* **Initial Value:** Setting a starting point for the sequence.
* **Increment:** Specifying the step size between generated values.
* **Cache Size:** Configuring how many values to pre-allocate for better performance.

1. GenerationType.**TABLE** –> Maintains a separate table to store and manage primary key values.Not frequently used due to its overhead and complexity.

@Entity annotation is a marker annotation used to discover persistent entities

For date time use @Temporal

*@Column*(name = "birth\_date")

*@Temporal*(TemporalType.DATE)

*private* Date birthDate;

Enum

*@Enumerated*(EnumType.STRING)

*@Column*(name = "payment\_status", nullable = *false*)

*private* PaymentStatus paymentStatus;

Composite Primary key

We need to implement this using a separate class using @Embeddable annotation

*@Embeddable*

*class* UserCompositePk *implements* Serializable {

*private* String name;

*private long* id;

}

And refer it in main Entity class using @EmbeddedId annotation

*@Entity*

*@Table*(name = "corporate\_user")

*class* User {

*@Id*

*@GeneratedValue*(strategy = GenerationType.AUTO)

*private int* id;

*@Column*(name = "first\_name")

*private* String firstName;

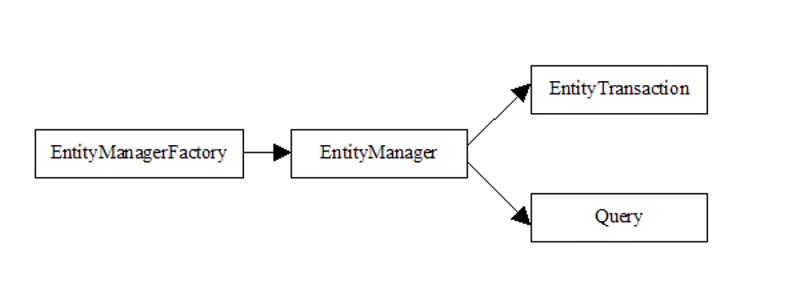
*@EmbeddedId*

*private* UserCompositePk userCompositePk;

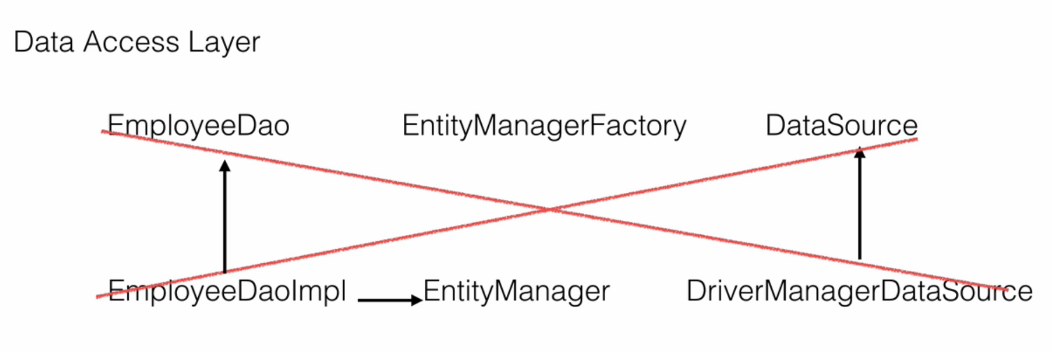
}

# Entity Manager

A connection to a database is represented by an **EntityManager** instance, which also provides functionality for performing operations on a database. There is a special object type **EntityManagerFactory** that is used to manage instantiation of EntityManager instances. Operations that modify the content of a database, require active transactions and are managed by an **EntityTransaction** instance obtained from the EntityManager. An EntiyManager instance also functions as a factory for query instances which are needed for executing queries on the database.



Spring Data removes all these boiler plate code



# Finder Methods

Spring boot JPA provides some finder methods to do queries

<https://docs.spring.io/spring-data/jpa/reference/repositories/query-keywords-reference.html>

|  |  |  |
| --- | --- | --- |
| fName | findByfName | Find exact match |
| lastName | findByLastNameContaining | Can search in case insensitive ways and search a part also. For e.g. for “Mr Violet Girl” can search with “Violet”, “violet” or “r violet”. But can’t search with “rviolet”. |
| fName | findByfNameContains | Same as above |
| remarks | findByRemarksLike | Like operator. For e.g. “Where’s the poop?” can be searched with “%poop%” or “%poop?” |
| schoolMarksPercentage | findBySchoolMarksPercentageBetween | http://localhost:8080/findBySchoolMarksPercentageBetween?marks1=90&marks2=60 |
| schoolMarksPercentage | findBySchoolMarksPercentageGreaterThan | http://localhost:8080/findBySchoolMarksGreaterThan?marks=90 |
| specialization | findBySpecializationIn | http://localhost:8080/findBySpecializationIn?specialization=MECHANICAL&specialization=ELECTRICAL |
| fName  lastName  remarks | findByfNameContainingAndLastNameAndRemarksContaining |  |
| remarks  Specialization  schoolMarksPercentage | findByRemarksContainingAndSpecializationInAndSchoolMarksPercentageBetween | http://localhost:8080/findByRemarksContainingAndSpecializationInAndSchoolMarksPercentageBetween?remarks=story&specialization=ELECTRICAL&specialization=IT&specialization=CSE&marks1=60&marks2=90 |

# JPA Mapping

One to One

Customer

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| [PROFILE\_UUID](http://localhost:8080/h2-console/query.do?jsessionid=ded69aafbb1ec64f76f3e930bfd757b0) | [CITY](http://localhost:8080/h2-console/query.do?jsessionid=ded69aafbb1ec64f76f3e930bfd757b0) | [CUSTOMER\_NAME](http://localhost:8080/h2-console/query.do?jsessionid=ded69aafbb1ec64f76f3e930bfd757b0) | [EMAIL](http://localhost:8080/h2-console/query.do?jsessionid=ded69aafbb1ec64f76f3e930bfd757b0) | [PHONE\_NUMBER](http://localhost:8080/h2-console/query.do?jsessionid=ded69aafbb1ec64f76f3e930bfd757b0) | [STATE](http://localhost:8080/h2-console/query.do?jsessionid=ded69aafbb1ec64f76f3e930bfd757b0) | [STREET](http://localhost:8080/h2-console/query.do?jsessionid=ded69aafbb1ec64f76f3e930bfd757b0) | [ZIP\_CODE](http://localhost:8080/h2-console/query.do?jsessionid=ded69aafbb1ec64f76f3e930bfd757b0) |
| 1 | Bartonchester | Rhea Curran | elvis.shanahan@yahoo.com | 1-795-275-4104 | New Jersey | 6445 Kunze Haven | 71097 |
| 2 | Elmoton | Ken Dahl | bonita.ohara@yahoo.com | 1-804-420-1339 | Arkansas | 450 Brekke Crest | 04052 |

*@Id*

*@GeneratedValue*(strategy = GenerationType.IDENTITY)

*private* Long customerUuid;

Profile

|  |  |
| --- | --- |
| [CUSTOMER\_UUID](http://localhost:8080/h2-console/query.do?jsessionid=ded69aafbb1ec64f76f3e930bfd757b0) | [NAME](http://localhost:8080/h2-console/query.do?jsessionid=ded69aafbb1ec64f76f3e930bfd757b0) |
| 1 | Rhea Curran |
| 2 | Ken Dahl |

Here customer and profile both are mapped with same key customer\_uuid in Customer is the PK as well as FK referencing Profile table

# Inheritance

We use inheritance in Java for achieving extensibility and reusability to design new classes using existing classes our domain classes are plain old Java objects which can use inheritance where applied

For example here we have a payment use case where we can make a payment using a card or a check. We have two options in this case card and check and when we apply inheritance to this all the common fields like ID and amount will go into a parent class called payment and the two modes of payment will have their own child classes which will extend this payment and they will have the specific fields or fields which are not common across. That is, a card will have a card number and a check will have a check number. We can have more fields like expiry date first name last name but they are ignored for simplicity reasons. But the underlying databases do not support inheritance mapping across the database tables. This is known as sub type problem in ORM.

And to solve this problem JPA provides inheritance mapping through three types of strategies

1. single table
2. table per class
3. joined.



## Single Table

Since it is going into one single table we need extra discriminator columns such as pMode we can call it anything but we need extra column that will differentiate between a card and a check so that hibernate when we save a record whether it's a card or a check.

Below is the table structure when credit card is saved the p\_mode value is cc when check is saved p\_mode is ch

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| id | p\_mode | amount | card\_number | check\_number |
| 0 | ch | 5687.000 | null | 435445456 |
| 1 | cc | 84870.000 | 7682-4526-9726-5258 | null |

On Java side abstract Payment class is created as Parent

*@Getter*

*@Setter*

*@AllArgsConstructor*

*@NoArgsConstructor*

*@Entity*

*@Inheritance*(strategy = InheritanceType.SINGLE\_TABLE)

*@DiscriminatorColumn*(name = "p\_mode", discriminatorType = DiscriminatorType.STRING)

*public abstract class* Payment {

*@Id*

*private int* id;

*private int* amount;

}

And child class Credit card and Check

*@Getter*

*@Setter*

*@AllArgsConstructor*

*@NoArgsConstructor*

*@Builder*

*@Entity*

*@DiscriminatorValue*("cc")

*public class* CreditCard *extends* Payment{

*private* String cardNumber;

}

*@Getter*

*@Setter*

*@AllArgsConstructor*

*@NoArgsConstructor*

*@Builder*

*@Entity*

*@DiscriminatorValue*("ch")

*public class* Check *extends* Payment{

*private* String checkNumber;

}

# Audit columns

@Entity

@Table(name = "user\_table")

public class User extends BaseEntity{

@MappedSuperclass

@EntityListeners(AuditingEntityListener.class)

public class BaseEntity {

@CreatedDate

@Column(updatable = false)

private LocalDateTime createdAt;

@CreatedBy

@Column(updatable = false)

private String createdBy;

@LastModifiedDate

@Column(insertable = false)

private LocalDateTime updatedAt;

@LastModifiedBy

@Column(insertable = false)

private String updatedBy;

@EnableJpaAuditing(auditorAwareRef = "auditAwareImpl")

@SpringBootApplication

public class IdentityServiceApplication implements CommandLineRunner {

@Component("auditAwareImpl")

public class AuditAwareImpl implements AuditorAware<String> {

/\*\*

\* Returns the current auditor of the application.

\*

\* @return the current auditor.

\*/

@Override

public Optional<String> getCurrentAuditor() {

return Optional.of("ACCOUNTS\_MS");

}

}

FetchType.EAGER can negatively impact performance in JPA under certain circumstances. Here’s why:

### 1. Loading Unnecessary Data

When using FetchType.EAGER, JPA will load the related entities immediately with the primary entity, even if the application doesn't need this related data at that moment. This can result in fetching a lot of unnecessary data, which can be inefficient and lead to increased memory consumption.

### 2. N+1 Select Problem

If an entity with EAGER fetched associations is loaded in a loop, it can cause the N+1 select problem. For example, if you load 10 customers, and each customer eagerly loads its orders, JPA might execute 1 query to load the customers and then N additional queries (one for each customer) to load their orders.

### 3. Complex Queries

Eager fetching can lead to very complex SQL queries, especially when multiple associations are eagerly fetched. This complexity can degrade performance and lead to difficulties in optimizing the database queries.

### 4. Serialization Issues

In web applications, entities are often serialized to JSON. Eagerly fetched associations can lead to large serialized objects, which can be inefficient and cause performance issues on the client side.

### Best Practices for Fetch Strategies

1. **Prefer FetchType.LAZY by Default**
   * Use FetchType.LAZY for most associations and load the necessary data on-demand using queries or by initializing the associations programmatically when needed.
2. **Use Fetch Joins in Queries**
   * When you know you will need related data, use JPQL or Criteria API with fetch joins to load the necessary associations efficiently in a single query.
3. **DTOs and Projections**
   * Instead of loading entire entity graphs, use DTOs or projections to fetch only the required data.
4. **Batch Fetching**
   * Configure batch fetching for collections and associations to reduce the number of queries.
5. **Limit Eager Fetching to Simple, Non-Recursive Associations**
   * If you need to use eager fetching, limit it to simple associations that don’t cascade into large object graphs.

### Common Performance Issues with Hibernate

1. **N+1 Select Problem**:
   * **Issue**: This occurs when Hibernate executes one query to retrieve a list of entities and then executes additional queries for each entity to retrieve related data.
   * **Solution**: Use fetching strategies like JOIN FETCH or entity graphs to retrieve related data in a single query.
2. **Lazy Loading Pitfalls**:
   * **Issue**: Lazy loading can lead to performance issues if not handled correctly, especially if it triggers multiple database queries in a loop.
   * **Solution**: Carefully manage lazy loading and consider using JOIN FETCH for relationships that are frequently accessed together. Monitor and optimize the number of queries generated.
3. **Excessive Flushes**:
   * **Issue**: Hibernate's automatic flush can lead to performance problems if it triggers database updates too frequently.
   * **Solution**: Control flushing behavior by using flush modes like FlushMode.COMMIT or FlushMode.MANUAL.
4. **Batch Processing**:
   * **Issue**: By default, Hibernate executes a separate SQL statement for each entity operation, which can be inefficient for bulk operations.
   * **Solution**: Enable batch processing by setting the hibernate.jdbc.batch\_size property and using Session.flush() and Session.clear() periodically to manage memory usage.
5. **Caching Issues**:
   * **Issue**: Ineffective use of first-level and second-level caches can lead to suboptimal performance.
   * **Solution**: Properly configure and use Hibernate's caching mechanisms. Use second-level cache providers like EHCache or Infinispan to reduce database access.
6. **Poorly Written Queries**:
   * **Issue**: Inefficient HQL (Hibernate Query Language) or JPQL (Java Persistence Query Language) queries can lead to performance bottlenecks.
   * **Solution**: Optimize queries, use indexes, and avoid fetching unnecessary data. Analyze the generated SQL and optimize it as needed.
7. **Large Result Sets**:
   * **Issue**: Fetching large result sets into memory can cause OutOfMemoryErrors and slow down the application.
   * **Solution**: Use pagination with setFirstResult and setMaxResults to limit the number of records retrieved. Consider streaming results if processing large datasets.
8. **Suboptimal Use of Collections**:
   * **Issue**: Improper handling of collections (e.g., List, Set) can lead to inefficient SQL generation.
   * **Solution**: Use appropriate collection types and consider using @BatchSize or @Fetch(FetchMode.SUBSELECT) annotations to optimize collection fetching.

### Best Practices for Improving Hibernate Performance

1. **Profiling and Monitoring**:
   * Regularly profile your application to identify performance bottlenecks. Tools like JProfiler, YourKit, or VisualVM can be useful.
   * Use Hibernate's built-in statistics and logging features to monitor query performance and cache usage.
2. **SQL Optimization**:
   * Analyze the SQL generated by Hibernate and ensure that it is optimized for your database. Use database-specific features and hints if necessary.
3. **Proper Indexing**:
   * Ensure that your database tables have the appropriate indexes to support your queries. This can significantly improve query performance.
4. **Minimize Object Retrieval**:
   * Retrieve only the necessary data. Avoid fetching entire entities when only a few fields are needed. Use projections or DTOs (Data Transfer Objects) if necessary.
5. **Session Management**:
   * Properly manage Hibernate sessions. Avoid long-running sessions that can hold onto database connections and memory.

### Conclusion

While Hibernate offers many advantages and is widely used in enterprise applications, it requires careful tuning and best practices to avoid performance issues. By understanding common pitfalls and implementing the suggested strategies, you can significantly improve the performance and scalability of your Hibernate-based applications.

spring.jpa.properties.hibernate.default\_batch\_fetch\_size=10

Setting spring.jpa.properties.hibernate.default\_batch\_fetch\_size=10 can improve performance by reducing the number of database queries needed to fetch associations, especially in scenarios where you have many entities with lazy-loaded associations. However, there are some potential drawbacks and considerations to keep in mind:

### Potential Drawbacks:

1. **Increased Memory Usage**:
   * Batch fetching loads multiple entities and their associations into memory at once. If the batch size is too large, it can lead to increased memory usage, which might cause issues in environments with limited memory.
2. **Complexity in Tuning**:
   * Finding the optimal batch size can be complex. Too small a batch size might not provide significant performance benefits, while too large a batch size can lead to memory issues and inefficient database access patterns.
3. **Impact on Database Load**:
   * While batch fetching reduces the number of queries, each query might become more complex and might take longer to execute. This can increase the load on the database, especially if the batch size is large and the database server is not optimized for handling large result sets.
4. **Potential for N+1 Problem**:
   * Although batch fetching can mitigate the N+1 select problem to some extent, it doesn't completely eliminate it. You might still encounter cases where the N+1 problem appears, albeit in a more controlled manner.

### Best Practices:

1. **Start with a Moderate Batch Size**:
   * Start with a moderate batch size, like 10 or 20, and monitor the performance. Adjust the batch size based on the performance metrics and memory usage.
2. **Monitor Performance and Memory Usage**:
   * Use monitoring tools to track the performance of your application and the memory usage. Adjust the batch size if you notice any performance degradation or high memory consumption.
3. **Load Testing**:
   * Conduct load testing with different batch sizes to understand the impact on your application’s performance and the database load. Choose a batch size that offers a good balance between performance and resource usage.
4. **Database Optimization**:
   * Ensure that your database is optimized for handling the queries generated by batch fetching. This might include indexing frequently accessed columns and optimizing the database schema.
5. **Use Fetch Profiles**:
   * Consider using Hibernate fetch profiles or entity graphs to control fetching strategies dynamically based on the use case. This provides more flexibility compared to a static batch fetch size configuration.

### Conclusion:

Setting spring.jpa.properties.hibernate.default\_batch\_fetch\_size=10 can provide performance benefits by reducing the number of database queries for fetching associations. However, it is essential to monitor the impact on memory usage and database load and adjust the batch size accordingly. By following best practices and tuning the batch size based on performance testing, you can achieve a balance that maximizes performance without causing resource issues.

In JPA (Java Persistence API), projections are a way to retrieve specific parts or subsets of an entity rather than the entire entity. This can be useful for improving performance and reducing the amount of data transferred between the database and the application. Projections allow you to select only the fields that you need, which can lead to more efficient queries.

There are three main types of projections in JPA:

1. **Entity Projections:**
   * Retrieve entire entities as they are mapped by JPA.
2. **Scalar Projections:**
   * Retrieve individual columns or fields from the database. This is similar to writing a SQL query that selects specific columns. Scalar projections allow you to select specific columns. You can use JPQL (Java Persistence Query Language) or a native SQL query to achieve this.
3. **DTO (Data Transfer Object) Projections:**
   * Retrieve data into a custom class (DTO) rather than an entity class. This is useful when you want to fetch specific fields and map them to a class that is not an entity.

Transactional

*@Transactional*

*public* Long insertCustomer(CustomerDto customerDto) {

*// Insert into Profile table*

String profileSql = "INSERT INTO Profile (email, name, phone\_number, street, city, state, zip\_code) VALUES (?, ?, ?, ?, ?, ?, ?)";

KeyHolder keyHolder = *new* GeneratedKeyHolder();

jdbcTemplate.update(connection -> {

PreparedStatement ps = connection.prepareStatement(profileSql, Statement.RETURN\_GENERATED\_KEYS);

ps.setString(1, customerDto.getEmail());

ps.setString(2, customerDto.getName());

ps.setString(3, customerDto.getPhoneNumber());

ps.setString(4, customerDto.getStreet());

ps.setString(5, customerDto.getCity());

ps.setString(6, customerDto.getState());

ps.setString(7, customerDto.getZipCode());

*return* ps;

}, keyHolder);

*// Retrieve the generated profile\_uuid*

Long profileUuid = Objects.requireNonNull(keyHolder.getKey()).longValue();

log.info("Profile UUID" +profileUuid);

*// Insert into Customer table using the generated profile\_uuid*

String customerSql = "INSERT INTO Customer (customer\_uuid, customer\_name) VALUES (?, ?)";

*int* customerUUid = jdbcTemplate.update(customerSql, profileUuid, customerDto.getName());

log.info("Customer UUID" +customerUUid);

*return* (*long*) profileUuid;

}

Other use case

The @Transactional annotation in Spring Boot is a powerful tool for managing transactions at the method or class level. Beyond the basic use case you presented, here are other scenarios where @Transactional can be useful:

### 1. Ensuring Atomicity Across Multiple Operations

In the example you shared, atomicity is achieved for multiple database operations. If any of the operations fail (e.g., inserting into the Profile table), the transaction rolls back, ensuring no partial updates occur.

### 2. Propagation Behavior

Learn about different propagation behaviors:

* **REQUIRED** (default): Reuses the current transaction or creates a new one if none exists.
* **REQUIRES\_NEW**: Suspends the current transaction and creates a new one.
* **NESTED**: Executes within a nested transaction.
* **MANDATORY**: Requires an existing transaction; throws an exception if none exists.
* Example: Nested service calls requiring different transaction behavior.

### 3. Read-Only Transactions

Use @Transactional(readOnly = true) for methods that only perform read operations. This can optimize performance by reducing lock contention or bypassing unnecessary persistence contexts.

java

Copy code

@Transactional(readOnly = true)

public List<Customer> getAllCustomers() {

return customerRepository.findAll();

}

### 4. Custom Rollback Rules

You can define specific exceptions that should trigger or suppress a rollback using the rollbackFor or noRollbackFor attributes.

java

Copy code

@Transactional(rollbackFor = CustomException.class)

public void updateCustomer(CustomerDto dto) {

// Logic to update customer

}

### 5. Using Transactions with Asynchronous Tasks

If you're using @Async methods, transactions won't propagate by default. However, combining @Transactional with @Async and ensuring proper thread configuration can help manage transactions in asynchronous workflows.

### 6. Using Transactions Across Multiple Databases

When working with multiple data sources, @Transactional can ensure consistency across distributed transactions. This typically requires a JtaTransactionManager or a tool like Spring's ChainedTransactionManager.

### 7. Handling Lazy Initialization Exceptions

Transactions help resolve LazyInitializationException when accessing lazy-loaded associations outside of the transactional context. You can ensure the required data is fetched within a transaction.

### 8. Retry on Optimistic Locking Failures

Use @Transactional in conjunction with retry mechanisms to handle optimistic locking failures caused by concurrent updates.

### 9. Declarative Transactions in Multiple Layers

Apply @Transactional at the service layer or repository layer, depending on your architecture:

* **Service Layer**: Orchestrates multiple repository operations.
* **Repository Layer**: Handles single-entity transactions.

### 10. Combining Programmatic and Declarative Transactions

Combine @Transactional with programmatic transaction management when dynamic transaction boundaries are required.

java

Copy code

TransactionTemplate transactionTemplate = new TransactionTemplate(transactionManager);

transactionTemplate.execute(status -> {

// Perform operations with finer control over commit/rollback

return result;

});

### Best Practices

* **Avoid Overuse**: Only use @Transactional where transactional boundaries are necessary.
* **Granularity**: Keep transaction scopes small to minimize locking and contention.
* **Exception Handling**: Be aware that unchecked exceptions (runtime exceptions) trigger rollbacks, while checked exceptions do not unless explicitly specified.

Multi Module Project

Say we want to make many modules and make one module dependent on another say other modules dependent on dto-repository



dto repository we need to define the group id artifact id and version

<parent>

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

<artifactId>spring-boot-starter-parent</artifactId>

<version>3.2.2</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<groupId>com.saha.amit</groupId>

<artifactId>dto-repository</artifactId>

<version>1.0-SNAPSHOT</version>

<packaging>jar</packaging>

<properties>

<maven.compiler.source>17</maven.compiler.source>

<maven.compiler.target>17</maven.compiler.target>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

<dependencies>

<dependency>

…

…

</dependencies>

<build>

<plugins>

<plugin>

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

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

And in the project we are using the dto-repository we add the dependency using group id artifact id and veriosn

<dependency>

<groupId>com.saha.amit</groupId>

<artifactId>dto-repository</artifactId>

<version>1.0-SNAPSHOT</version>

</dependency>

Spring Doc Swagger

<https://springdoc.org/>

<dependency>

<groupId>org.springdoc</groupId>

<artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>

<version>2.3.0</version>

</dependency>

Swagger will be available at <http://localhost:8080/swagger-ui/index.html>

Making a bean read only i.e it would not be available in update request body add

@JsonProperty(access = JsonProperty.Access.***READ\_ONLY***)

private int id;

Request body

{

"name": "string",

"email": "string",

"phoneNumber": "string",

"role": "string",

"password": "string"

}

Opposite make a variable available only in request body but not in Response

@JsonProperty(access = JsonProperty.Access.***WRITE\_ONLY***)

private String password;

{

"id": 0,

"name": "string",

"email": "string",

"phoneNumber": "string",

"role": "string"

}

API description

For the errors scenarios, what we have to do is that we have to Throw a custom exception and Catch that in controller advice and and then send the appropriate response and define that in Swagger documentation

@Operation(

summary = "Fetch User Details REST API",

description = "REST API to fetch User details based on a mobile number"

)

@ApiResponses({

@ApiResponse(

responseCode = "200",

description = "HTTP Status OK"

),

@ApiResponse(

responseCode = "500",

description = "HTTP Status Internal Server Error",

content = @Content(

schema = @Schema(implementation = ErrorResponseDto.class)

)

)

}

)

# Properties file











Setting environment variables from intellij



If we want to set properties in an Class or Record

@ConfigurationProperties(prefix = "accounts")

public record AccountsContactInfoDto(String message, Map<String, String> contactDetails, List<String> onCallSupport) {

}

@EnableConfigurationProperties(value = {AccountsContactInfoDto.class})

@SpringBootApplication

public class ProductServiceApplication implements CommandLineRunner {

api.info=Product API

accounts.message=Welcome to Buying and Selling product

accounts.contactDetails.name=John Doe - Developer

accounts.contactDetails.email=john@mailinator.com

accounts.onCallSupport=(555) 555-1234, (555) 523-1345