Lecture Notes: Introduction to Spring Core and Spring Boot

Why Spring?

1.1 Coupling

Coupling is the relationship among the different classes of any application. This relationship can be of two types – tight coupling and loose coupling. A loosely coupled application allows easy testing and debugging in case anything goes wrong. Also, in loosely coupled applications, it is easy to reuse or replace any component with any other components. However, if any tightly coupled application stops working, it is difficult to determine the underlying cause. Moreover, for tightly coupled applications, it is tough to replace any component of the application with some other component.

When multiple classes are highly dependent on each other, they are said to be 'tightly coupled'. In software development, normally, an object of one class needs an object of another class to execute its functionality. For example, a car needs an engine to execute its functionality.
When you refer to a dependency class directly inside the dependent class, you are introducing tight coupling.

```
class Car {
        Engine engine;
}
```

```
class Engine {

}
```

Here, the car class (dependent class) is tightly coupled with the engine class (dependency class). Tight coupling can also be understood from the code snippet given below.

```
public class GreetingService {
   public void greet(String name) {
      System.out.println("Hello, " + name);
   }
}
```

```
public class Main {
   public static void main(String[] args)
{
      GreetingService greetingService = new GreetingService();
greetingService.greet("John");
   }
```

```
}
```

Here, the Main class depends on the GreetingService class. When you create the object of the GreetingService class directly inside the Main class, you introduce a tight coupling between both the classes. Making changes in a tightly coupled code is tough and adding new features to such a system is also a complex process.

1.2 Loose Coupling

You can create loosely coupled applications by coding for the interfaces and by making use of the factory design pattern.

```
public interface GreetingService {
    public void greet(String name);
}
```

```
public class EnglishGreetingService implements GreetingService {
    @Override
    public void greet(String name) {
        System.out.println("Hello, " + name);
    }
}
```

```
public class FrenchGreetingService implements GreetingService {
    @Override
    public void greet(String name) {
        System.out.println("Bonjour, " +     name);
    }
}
```

```
public class GreetingServiceFactory {
    public GreetingService getGreetingService(String language) {
        if (language.equals("english")) {
            return new EnglishGreetingService();
        } else if (language.equals("french")) {
            return new FrenchGreetingService();
```

```
        } else {
            throw new RuntimeException("No greeting Service exist for
" + language + " language.");
        }
    }
}
```

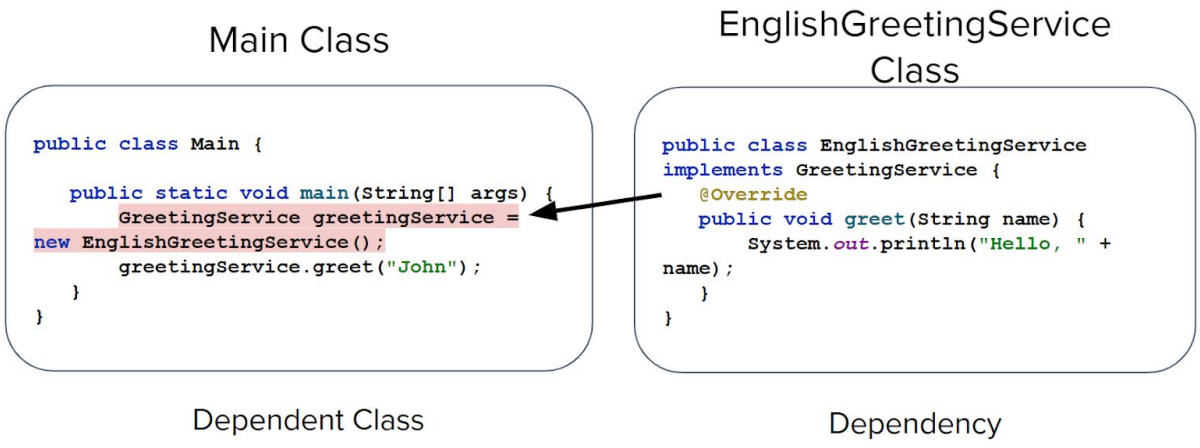
```
public class Main {

    public static void main(String[] args) {
        GreetingServiceFactory greetingServiceFactory = new
GreetingServiceFactory();
        GreetingService greetingService =
greetingServiceFactory.getGreetingService("french");
        greetingService.greet("John");
    }
}
```

You first create an interface GreetingService, and then, write its two implementations – EnglishGreetingService and FrenchGreetingService. Then, you write a factory class that will return the object of these classes depending on the language passed. Later, in the Main class, you create an object of this factory class and call the method present inside it, which, in turn, calls the greet() method. So, in this example, the Main class is not coupled tightly with the EnglishGreetingService class or the FrenchGreetingService class. You can also create more greeting services and use them in your Main class through GreetingServiceFactory without significantly changing the code in the Main class. It is a standard industry practice to use loose coupling when working on any enterprise-level application.

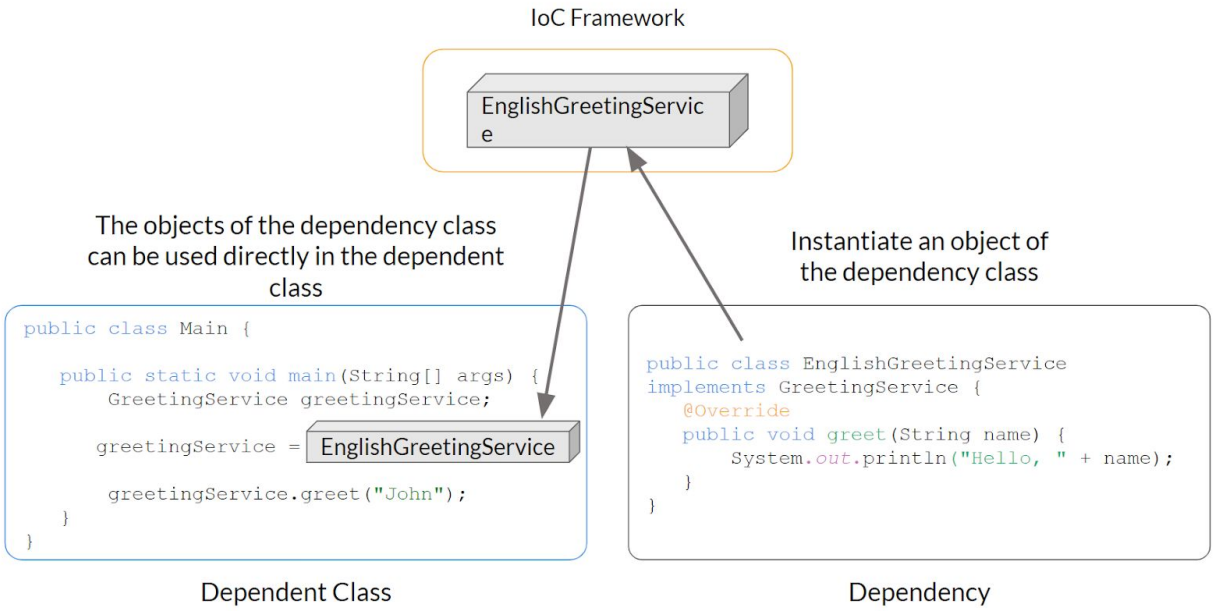
1.3 Inversion of Control (IoC)

In the normal code flow, the developer first creates a class, followed by its object, to invoke the methods present inside this class.



Normal Flow

However, when you start using frameworks that rely on the concept of IoC, these frameworks start taking care of the creation of the objects of the classes. Once a framework has created the object, it can then be used inside the class that depends on this object.



Inverted Flow

So, to sum it up, any framework that relies on the concept of IoC helps you create the object of the class. You just need to tell the framework that you need an object for a particular class.

1.4 Dependency Injection

A class might require objects of some other class. In such cases, the class that requires the object is said to be the dependent class and the class whose objects are required is called the dependency class. This is where the dependency injection (DI) helps you to inject dependencies into a class. IoC is a generic programming principle, whereas DI is an implementation of IoC.

IoC states that the dependent classes should not create objects of the dependency classes. Dependency objects would be created by the IoC container or Framework, and then, provided to the dependent object. IoC does not describe how dependency objects will be provided to the dependent class.

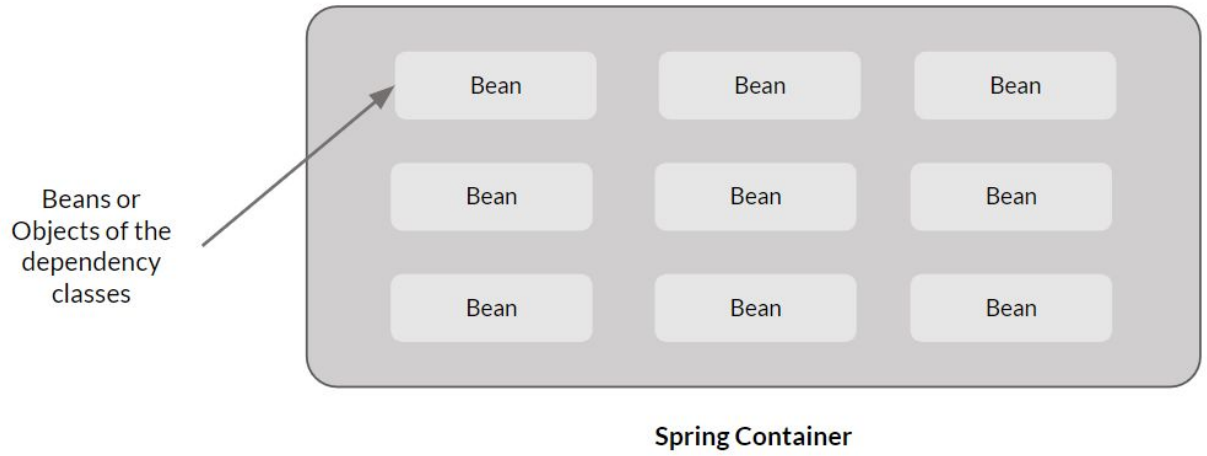
To summarise, the framework built on the principle of IoC takes care of all the dependencies of a dependent class and provides it with the objects of all the dependency classes. This saves a lot of time and effort of a developer.

1.5 Introduction to Spring

Previously, you learnt to make applications loosely coupled by writing extra code in the form of the factory pattern. For big enterprise applications, this boilerplate code can get quite complex and divert you from the application's main goal. This is where the Spring Framework helps you by providing this boilerplate code.

A framework, or a software framework, provides you with a well-tested generic boilerplate code written by some of the best software developers by following design principles and best practices. This will not only save time but also insert the correct components in your applications. This is a fast and easy approach. Almost all frameworks, including Spring, rely on the concept of Inversion of Control (IoC).

The IoC Framework creates the objects of the dependency classes that can be used by the dependent classes. These frameworks behave like containers for such objects and are also called IoC containers. The IoC container of the Spring Framework is called the Spring Container. In Spring, the dependency objects contained inside the Spring Container are called Spring beans, or simply, beans.



Spring provides two types of containers:

- Spring BeanFactory Container
- Spring ApplicationContext Container

BeanFactory is similar to a factory class that contains a collection of methods to get beans from the Spring Container. It is lightweight, as it instantiates beans only when requested by the application instead of creating all the beans at the start of the application. Being lightweight, it is generally used for mobile device applications.

ApplicationContext is an interface that extends the BeanFactory interface. ApplicationContext is a heavyweight interface compared to BeanFactory, as it loads all the beans at the startup of the application. It provides some extra enterprise-specific functionalities on top of the BeanFactory container. Since it includes all the functionalities of the BeanFactory container and also provides some additional facilities, it is generally recommended over BeanFactory wherein the startup time is not an issue, such as when writing the server-side code.

Features of Spring framework:

- It is open-source.
- It is lightweight in terms of execution, as it does not require an external container support to run the applications, such as the Servlet container that is used to run the Servlet/JSP application, and because of its POJO implementation.
- Modular: It is developed in different modules. So, you can import only those Spring modules that are needed as per the requirements of the application and ignore the rest.
- The integration with other frameworks is loosely coupled, such as integration with the Hibernate Framework.
- Aspect-oriented programming (AOP): It uses AOP to separate cross-cutting concerns (logging, security, exception handling, etc.) from the business logic.
- Dependency injection: It helps in developing loosely coupled applications, and, in turn, facilitates easier unit testing.
- IoC containers: It creates and manages the lifecycle of the Spring beans.
- Transaction management: It is used to maintain transactions and is mostly used to maintain database transactions.

Spring Core

2.1 Creating a Maven Project

For developing any big enterprise applications, you may need hundreds of jars. Downloading these hundreds of jars from different websites and adding them to the classpath can be quite tedious.

The external jars are also called API plugins, a software component that adds additional features to existing applications. A jar is a set of Java classes, interface, enums and annotations that are zipped into a .jar file. Jar files are mostly provided by third-party vendor companies. They need to be added separately into the application classpath. Some of the examples of API plugins are as follows:

- The Spring Framework
- JDBC Type 4 Driver

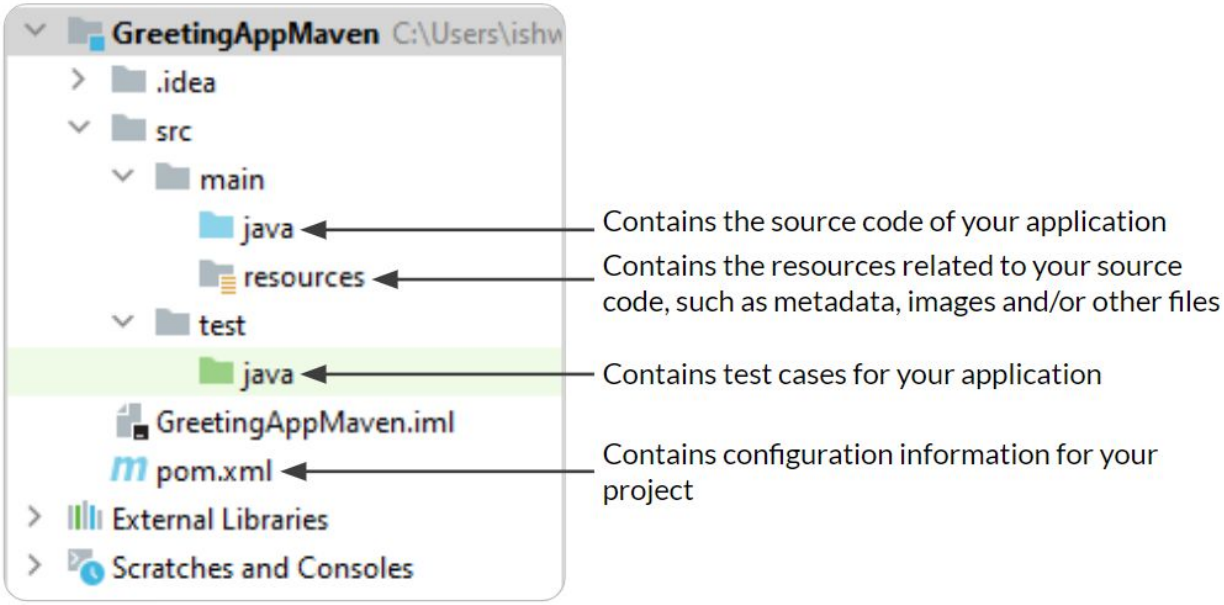
This is where Maven comes into play. You only need to provide a list of dependencies needed for your application to the Maven, and it will download and add them to the classpath.

Maven is a software application build tool. It is primarily used to download .jar files. Other uses of Maven are as follows:

- It provides a standard project structure.
- It helps you in compiling, building and deploying a software.
- It also helps you in generating the documentation for the source code.

Maven repositories refer to a location where all Maven dependencies are stored. If you start working for some big organisation, then the organisation creates a repository after performing security checks and puts all the different .jar files in that repository.

pom.xml file contains all the project-related configuration information that will be used by Maven to build an application. It also contains a list of all the API plugin jar dependencies that need to be downloaded. It resides in the base directory of a project.



Maven Project Structure

2.2 Bean Creation

Developers write dependency classes. So, they need to tell the Spring about the classes that should be treated as dependency classes. They can provide this information to the Spring in the form of metadata (information about your code). There are three ways to provide metadata to the Spring, which are as follows:

- **XML-based configuration**: You provide an XML file in which all the dependency classes are listed.

- **Java-based configuration**: You provide a type of factory class that contains the methods that return the objects of the dependency classes. The Spring Container executes these methods to instantiate dependency classes. This factory class needs to be marked with the @Configuration annotation.
- **Annotation-based configuration**: Annotations are a way to provide metadata about your code. Annotations are similar to comments, not for other developers, but for the tools that process your source code. Here, that tool is the Spring Framework. Annotation-based configuration helps you provide a shorter and a more concise configuration. To create a bean for a class, you need to mark it with @Component annotation.

```
@Component
public class EnglishGreetingService implements GreetingService {
    @Override
    public void greet(String name) {
        System.out.println("Hello, " + name);
    }
}
```

To create a bean for the EnglishGreetingService class, you need to mark the EnglishGreetingService class with the @Component annotation. The Spring Container will create a bean by instantiating the EnglishGreetingService class. The name of the bean would be the same as the name of the class, with the first letter in lower case. Thus, the bean's name here would be 'englishGreetingService'.

To get the bean out of the container, you can make use of the ApplicationContext, and then, you can invoke the different methods defined in that class.

2.3 Dependency Injection

When you mark classes with the @Component annotation and when the Spring Container is loading at the start of the application, it will contain beans for those classes. But, what if beans are themselves dependent on other beans? How can you instruct the Spring Container to inject one bean into another?

This is where the dependency injection (DI) comes into play. It helps you to inject dependencies into an object. DI is one of the implementations or a subtype of IoC and instructs how to provide dependency objects to the dependent objects. DI states that dependencies can be injected into the dependent objects using the setter method or a constructor. In Spring, DI is implemented using the @Autowired annotation. You can implement DI in the following three ways:

- Field or property injection
- Setter injection

- Constructor injection

Based on these three ways, the @Autowired annotation can be applied over the following:

- Fields or properties
- Setter methods
- Constructors

```
@Component
public class EnglishGreetingService implements GreetingService {

    @Autowired
    private TimeService timeService;

    @Override
    public void greet(String name) {
        //Implementation goes here
    }
}
```

In field-based injection, you place the @Autowired annotation over the field or the property.

```
@Component
public class EnglishGreetingService implements GreetingService {

    private TimeService timeService;

    @Autowired
    public void setTimeService(TimeService timeService) {
        this.timeService = timeService;
    }

    @Override
    public void greet(String name) {
        //Implementation goes here
    }
}
```

In setter-based injection, you place the annotation over the setter methods.

```
@Component
public class EnglishGreetingService implements GreetingService {

    private TimeService timeService;

    @Autowired
    public EnglishGreetingService(TimeService timeService) {
        this.timeService = timeService;
    }

    @Override
    public void greet(String name) {
        //Implementation goes here
    }
}
```

In constructor-based annotation, you place the @Autowired annotation over the constructor.

The property and the setter injections should be used to inject the dependencies that are not critical for the dependent object to perform its functionalities and the dependencies that may change over the course of time.

The constructor injection is used to inject the dependencies that are critical for the dependent object to perform its functionalities.

2.4 Resolving Conflicts

When more than one bean of the requested type is present, the Spring Container will throw the NoUniqueBeanDefinitionException exception. How to instruct the Spring about the bean that is to be injected into the greeting service? This is where the @Qualifier annotation helps specify the bean that is to be injected using the bean name.

The code snippet for property injection is as follows:

```
@Autowired
@Qualifier("timeService24HourFormat")
private TimeService timeService;
```

The code snippet for setter injection is as follows:

```
@Autowired
@Qualifier("timeService24HourFormat")
public void setTimeService(TimeService timeService) {
   this.timeService = timeService;
}
```

The code snippet for constructor injection is as follows:

```
@Autowired
public EnglishGreetingService(@Qualifier("timeService24HourFormat")
TimeService timeService) {
   this.timeService = timeService;
}
```

2.5 Bean Scope

When you mark a class with the @Component annotation, the Spring Container will treat that class as a dependency and instantiate that class to create a bean. By default, Spring creates one bean per dependency class, but you can control that setting using the @Scope annotation.

The Spring framework supports the following five types of bean scopes:

- Singleton
- Prototype
- Request
- Session
- Global-session

Out of these, the last three types of scopes are applicable only in the case of a web-aware ApplicationContext.

Singleton is the default scope of Spring bean inside the Spring IoC container. It is best described as One Bean Per @Component Class Per IoC Container. The Spring IoC container gives the same bean reference every time when a request is made for a bean using the getBean() method or when a bean needs to be injected using the @Autowired annotation. This scope is used for stateless beans (which do not maintain any state).

When the bean scope is set to Prototype, the Spring IoC container gives a new bean instance reference every time a request is made for a bean using the getBean() method or when a bean needs to be injected using the @Autowired annotation. Thus, a single bean definition of the Prototype scope can have any number of object instances. This scope is used for stateful beans (that maintain the state).

The remaining three scopes are available only for web-based applications, which are as follows:

- request: The Spring container gives a single bean instance per HTTP request.
- session: The Spring container gives a single bean instance per HTTP session (user-level session).
- global-session: The Spring container gives a single bean instance per global HTTP session (application-level session).

Spring Boot

3.1 Introduction to Spring Boot

Spring helps you to create loosely coupled applications without the need to write the boilerplate code. But when you are developing big enterprise applications using Spring, you have to use a lot of external libraries and provide configurations to create beans for them. This compels you to write a lot of 'boilerplate configurations'.

This is where Spring Boot comes to the rescue. It eliminates the need to provide boilerplate configuration for each and every external library used by you. So, Spring eliminates the need for 'boilerplate code for loosely coupled applications', whereas Spring Boot eliminates the need for 'boilerplate configurations for Spring applications'. With Spring Boot, you need to focus only on the business logic, without thinking about boilerplate code and boilerplate configurations.

What is Spring Boot?

Spring Boot is one of the sub-projects of the Spring Framework, which eliminates the need to provide boilerplate configurations.

How does it eliminate the need to provide configurations?

It takes an Opinionated view of building Spring applications. This means that Spring Boot comes with certain default configurations, and it makes some smart assumptions to auto-configure your application. These default configurations are generally the ones that are popularly used and help in easier and faster development of applications. All of these defaults can be easily overridden, although they are mostly preferred in their default state. It is like the autocomplete feature used by Google search.

For example, if you put a JDBC (Java Database Connectivity) dependency in your pom.xml file, then it will download all other related dependencies, set up an in-memory database for you and create a datasource bean, which will be injected automatically into your application. Another example of auto-configuration is the Tomcat server. Spring Boot comes with Tomcat as an embedded server. Tomcat is one of the most widely used web application servers. So, with Spring Boot, you have to provide minimal configuration to run your application.

Spring vs. Spring Boot

Spring	Spring Boot
Eliminates the need to write boilerplate code	Eliminates the need to write boilerplate code and boilerplate configuration
You need to list down each and every dependency separately in the pom.xml file.	You only need to list down the starter projects in the pom.xml file. These starter projects will automatically download the required dependencies.
It takes time and effort to run Spring projects as you need to provide a lot of configurations.	You can run Spring Boot projects as soon as you create one.
It requires additional support to set up the server.	It comes with embedded servers, such as Tomcat or Jetty, which are fully configured.

Spring Boot - Features

- **Auto-configuration:** Based on the dependencies added in the pom.xml file, Spring Boot implements the necessary configurations automatically.
- **Standalone:** Web applications that are built with Spring Boot do not need to be deployed on a web server. You just need to click the run button and Spring Boot will use the embedded server to run the application.
- **Opinionated:** Spring Boot uses default configurations to make smart assumptions to auto-configure your application. This reduces development time and improves efficiency.
- **Integration with Other Frameworks:** It is very easy to integrate Spring Boot applications with other frameworks, such as Spring Data, Spring JDBC and Spring Security, since Spring Boot uses opinionated defaults to configure them.
- **Starter Project:** With Spring Boot, you only need to specify starter projects in the pom.xml file and it will download all the dependencies and configure them itself. Thus, you do not need to list down each and every dependency individually.
- **Logging and Security:** Applications that use Spring Boot are provided proper security and logging so you can keep track of them as well as their development.

3.2 Creating a Spring Boot Project

There are four methods to develop a Spring Boot application using Maven as shown below:

1. Spring Boot Initializr
2. Spring Starter Project Wizard Of IntelliJ or Eclipse
3. Spring Boot CLI (a command-line interface to build Spring Boot applications using Groovy)
4. Spring Maven Project

Of these four methods, Spring Boot Initializr is most commonly used to develop Spring Boot applications.

Spring Boot Starters

Spring Boot Starters contain a list of all related dependency jars, which are used together to implement a type of feature. Thus, instead of adding Maven dependencies for each jar, you can just mention the starter POM using the <dependency> tag and it will help you download all the related dependencies into the project. All the starters are named in a similar pattern, which is spring-boot-starter-*, where * is the type of application.

For example, when you add spring-boot-starter-web into your pom.xml file, it will download all the dependencies that are required to build a RESTful web application using Spring Boot, such as Spring MVC, Tomcat server and hibernate-validator.

Following are some of the commonly used starters:

- spring-boot-starter: Core starter. Used to enable logging, auto-configuration and YAML
- spring-boot-starter-web: Used for creating web and RESTful applications with Spring MVC
- spring-boot-starter-data-jpa: Used for database access using Spring Data JPA with Hibernate
- spring-boot-starter-security: Used for Spring Security
- spring-boot-starter-test: Used for testing an application with libraries such as Junit and Mockito
- spring-boot-starter-aop: Used for Aspect-Oriented Programming with Spring AOP and AspectJ

Spring Boot Dependency Management

When you are building a Spring Boot Project, you do not specify the versions of the dependencies or the starter POMs in the project's pom.xml file. You only specify the version of spring-boot-starter-parent and simply add the dependencies into the pom.xml file. Spring Boot will decide suitable versions of the dependencies and download them. When you change the version of spring-boot-starter-parent, Spring Boot will upgrade all the dependencies automatically. However, if you still want to specify the version of the dependency, then you can do so in the pom.xml file.

3.3 Embedded Servers and Auto Configuration

Embedded Servers

With Spring Boot, web applications can be packaged as a .jar file. This file contains both the application code and an embedded server (servers that are embedded as part of the deployable

application). Thus, with Spring Boot, you can build web applications in just two steps as shown below:

- Develop the web application
- Click the 'run' button (just like a normal Java application)

Spring Boot supports three embedded servers/containers for application building: Tomcat, Undertow and Jetty. Tomcat is the (Opinionated) default embedded server for Spring Boot applications. Nevertheless, you can configure your application to have Jetty or Undertow as the default embedded server through the pom.xml file.

Spring Boot Auto-Configuration

Normally, auto-configuration is triggered by annotating the Main class using the @SpringBootApplication annotation. This annotation is a combination of the following three annotations:

- @SpringBootConfiguration: It enables you to provide a Java-based configuration in the Main class.
- @EnableAutoConfiguration: It triggers auto-configuration.
- @ComponentScan: It enables you to scan the @Component classes, which are present in the package that contains the Main class or its subpackages.

3.4 Web Applications

Web application refers to an application that is installed on a remote machine and hosted using web servers, such as Apache Tomcat and Jetty. It is accessed via a public network called the Internet through the HTTP protocol. End users can use a web browser to access the web application. As web applications are hosted on servers and accessed via web browsers, they consist of two segments:

- Back-end segment: This code runs on the server and interacts with the database.
- Front-end segment: This code controls how the web pages will be rendered in the browser.

Back-end segment is the part of a web application that runs on the web servers and interacts with the database. It accepts requests from the browsers, processes those requests using business logic, stores and fetches data from the database, prepares a response, and sends the response back to the front-end. It is designed/developed by back-end developers using server-side technologies, such as Java and PHP, or frameworks, such as Spring and Node.

Front-end segment consists of the Graphical User Interface (GUI) and is responsible for all the UI-related functional work. It helps users to send a request to the back-end using GUI components and render the response from the back-end on the web browser. It is designed/developed by front-end developers using client-side technologies, such as HTML, CSS and JavaScript, or frameworks, such as Angular and React.

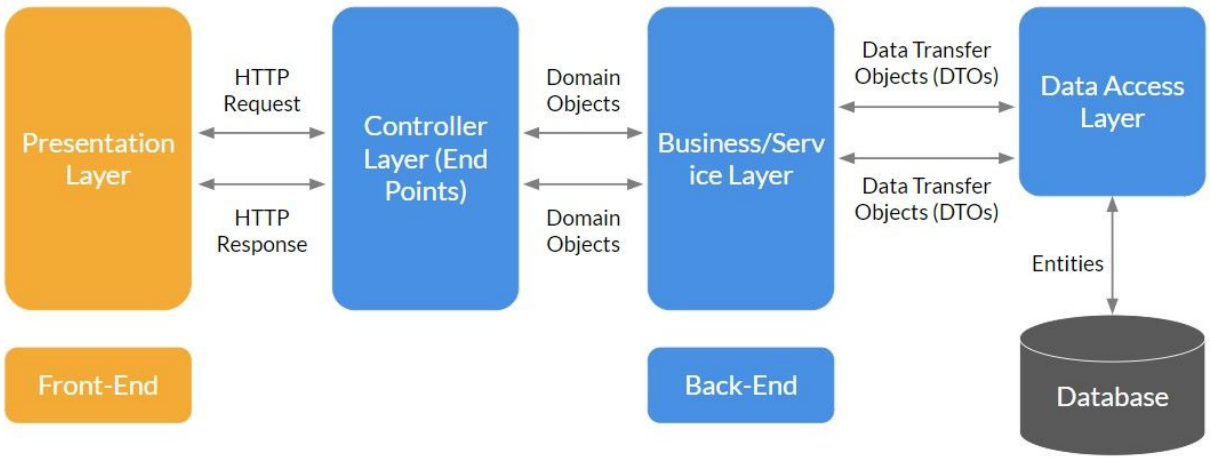
Layered Architecture

Layered Architecture is a technique that helps you in building loosely coupled applications by breaking down an application into layers and organising its different components into different layers based on the roles and responsibilities of those components. All these layers (or, the components inside the layers) will be connected to each other in a loosely coupled manner using the concepts of Inversion of Control (IoC) and Dependency Injection (DI).

Typically, a web application is distributed into three layers:

- Data access layer
- Service/Business layer
- Presentation/Controller layer

Each layer has a separate role and contributes differently to the application.



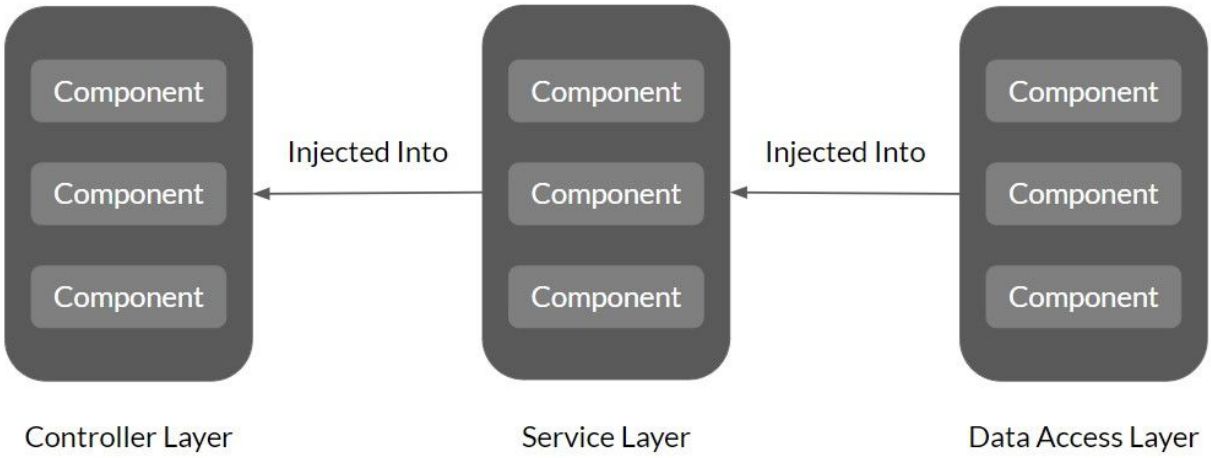
Layered Architecture

Data access layer is a part of the back-end segment. All the components that are responsible for interacting with the database will be a part of this layer. The components inside this layer will not be dependent on the components of the other two layers.

Service layer is also a part of the back-end segment. It consists of all the components that are responsible for processing the client request based on the business logic and for preparing the response. The components inside this layer use the components of the data access layer to process the client request and prepare the response. Thus, the components of the data access layer will be injected into the components of the service layer using the concepts of IoC and DI.

Presentation layer is a part of both the front-end and the back-end. In the back-end, it consists of the controller layer or the end points, which are used by the front-end to send the request to the back-end and wait for the response. This layer will consist of components that expose the end points to the front-end. The components inside this layer use the components of the service layer to process the request and generate the response. Thus, the components of the

service layer will be injected into the components of the controller layer using the concepts of IoC and DI.



Dependency Flow in a Layered Architecture

3.5 Configuring a Spring Boot Project

While creating beans, Spring Boot first checks whether the developer has provided custom properties for that bean in the application.properties file. By default, this file is empty, which implies that Spring Boot uses only the opinionated defaults for the auto-configuration of beans. For example, if Spring Boot finds a Tomcat dependency in the classpath, then it will configure and set up the server for you. By default, the server will run on port 8080. But you can change this setting by defining the following property in the application.properties file: server.port=8081.

The application.properties file is one of the means of providing custom properties to override the opinionated defaults used by Spring Boot.

Some alternatives, based on the order of precedence, are as follows:

- Command-line arguments
- Java system properties
- Environment variables
- YAML files

With these tools, you can externalise your configuration such that the same application code can work in different environments.

Configuration using Spring Profiles

An application goes through multiple stages during development, such as ‘dev’, ‘testing’ and ‘prod’. At different stages, you would want to configure your application differently. One way to

do this is to modify the custom configuration in the application.properties every time the application moves from one stage to another. But this approach is quite cumbersome and error-prone. This is where Profiles come to the rescue. Spring Profiles help you provide environment-specific properties for your applications.

You can provide environment-specific custom configurations in the application-{profile}.properties file. So, there can be multiple properties files, one for each environment, and a single properties file for the default settings. Following are some examples of such files:

- application.properties
- application-dev.properties
- application-testing.properties
- Application-prod.properties

You can also control bean creation for different profiles by marking a component class with the @Profile("profile-name") annotation.

You can set active profiles in one of the following ways:

- Java System Properties:
  -Dspring.profiles.active=dev
- Environment Variables:
  export spring_profiles_active=dev
- Properties Files:
  spring.profiles.active=dev