# Introduction to Spring Core and Spring Boot

**Session:** Spring Boot
**Instructor:** Vishwa Mohan

# Spring Boot

# Introduction to Spring Boot

# SPRING BOOT


- Previously, you learnt how Spring helps you to **create loosely coupled applications** without the need to write **boilerplate code**.
- When developing big enterprise applications using Spring, you have to use a number of **external libraries** and **provide configurations** to create beans for them.
- You have to write a number of '**boilerplate configurations**'.

# SPRING BOOT

- Spring Boot comes to the rescue and **eliminates** the need to **provide boilerplate configuration**.
- 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**.

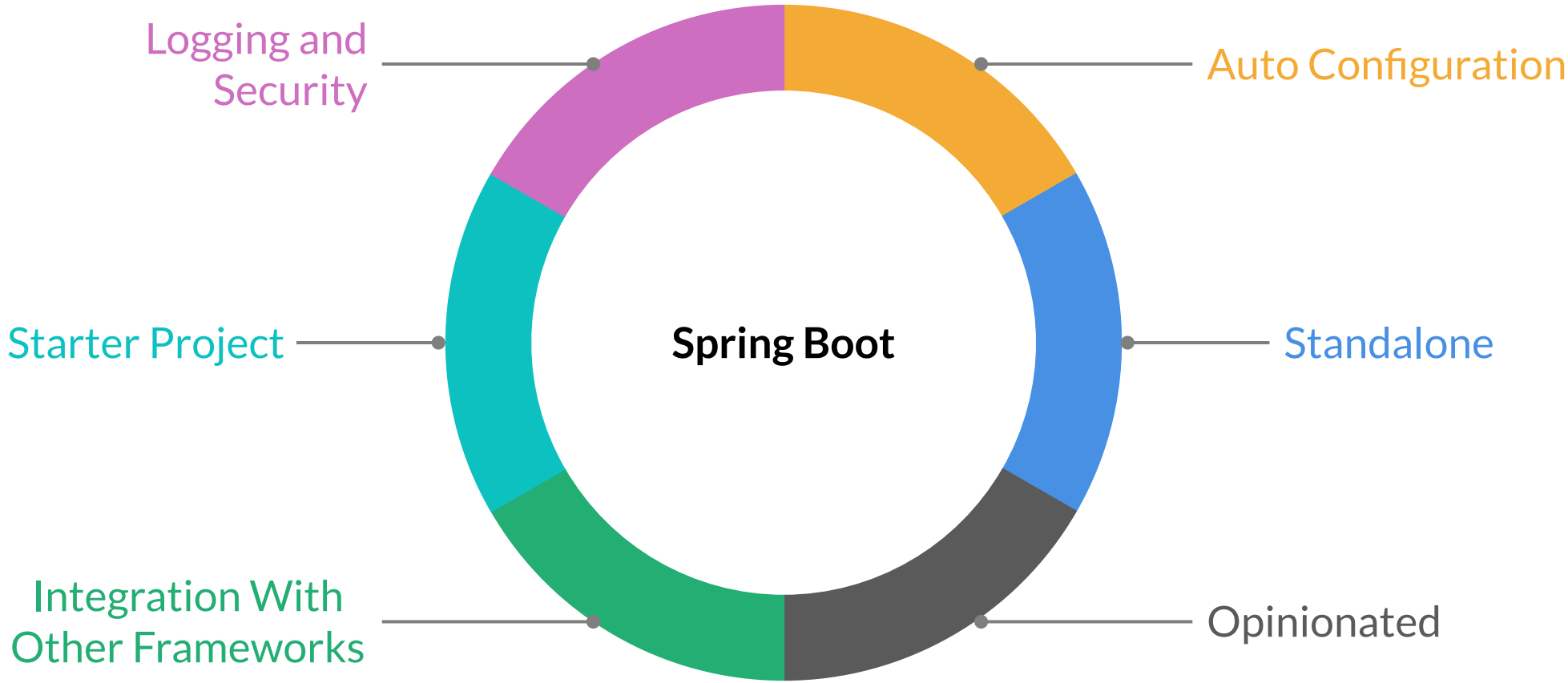
# WHAT IS SPRING BOOT?

- Spring Boot is one of the sub-projects of the Spring Framework, which eliminates the need to provide boilerplate configurations.
- It takes an **Opinionated view** of building Spring applications. 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 used popularly and help in **easy** and **faster development** of applications.
- It is like the **autocomplete feature** used by Google search.

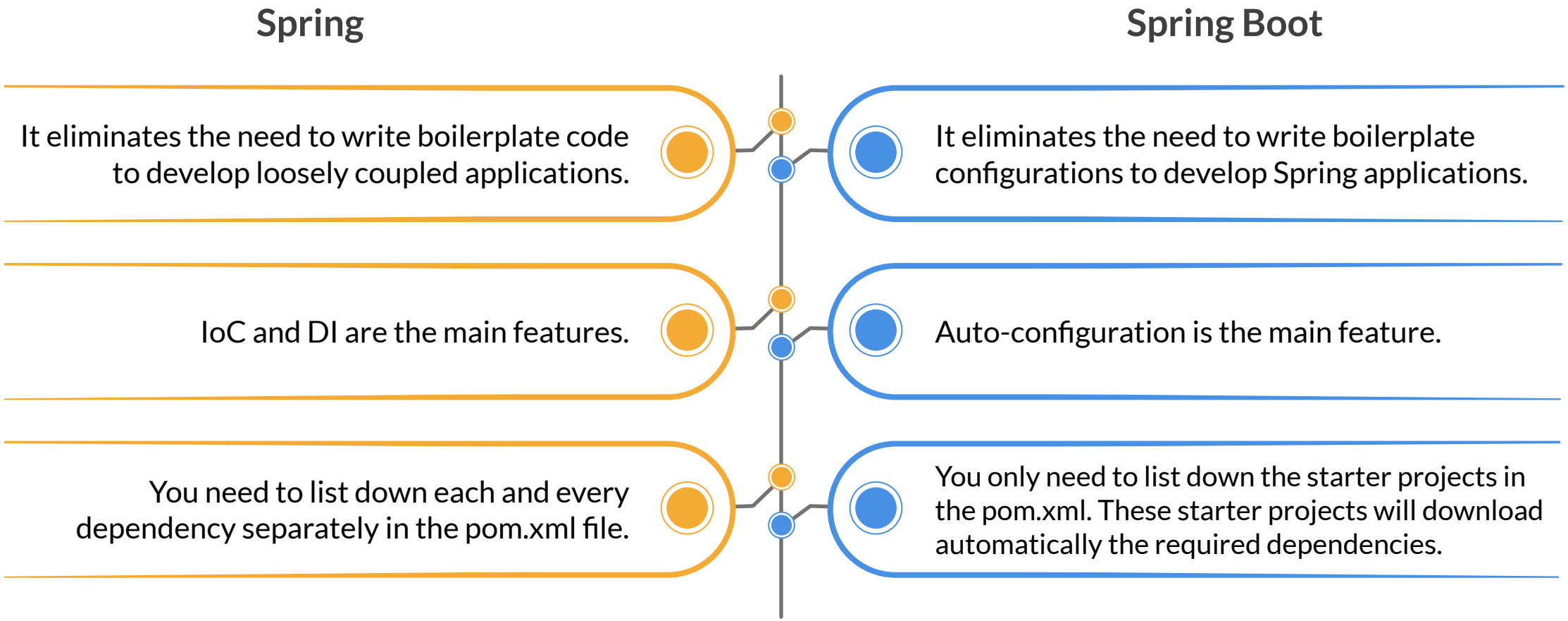
# WHAT IS SPRING BOOT?

- For example, if you put a Java Database Connectivity (JDBC) 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 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 get your application running.

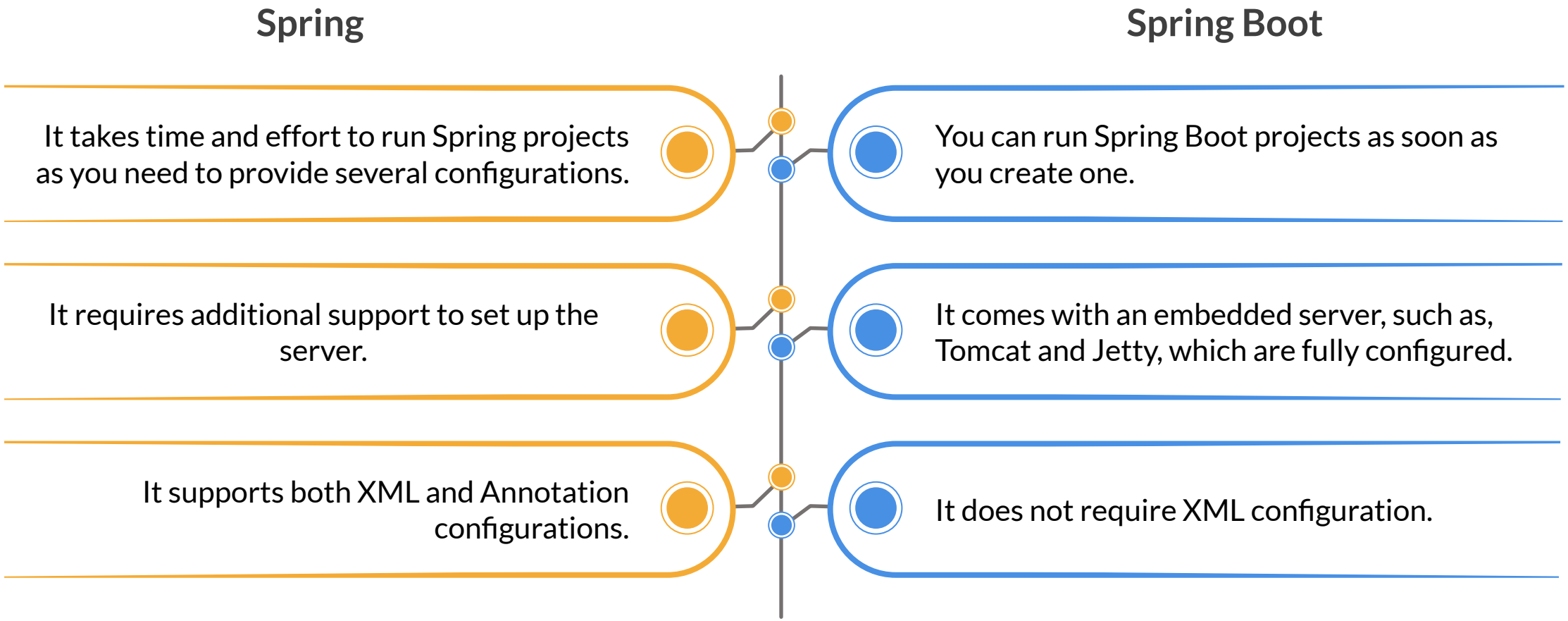
# SPRING BOOT FEATURES



# SPRING VERSUS SPRING BOOT

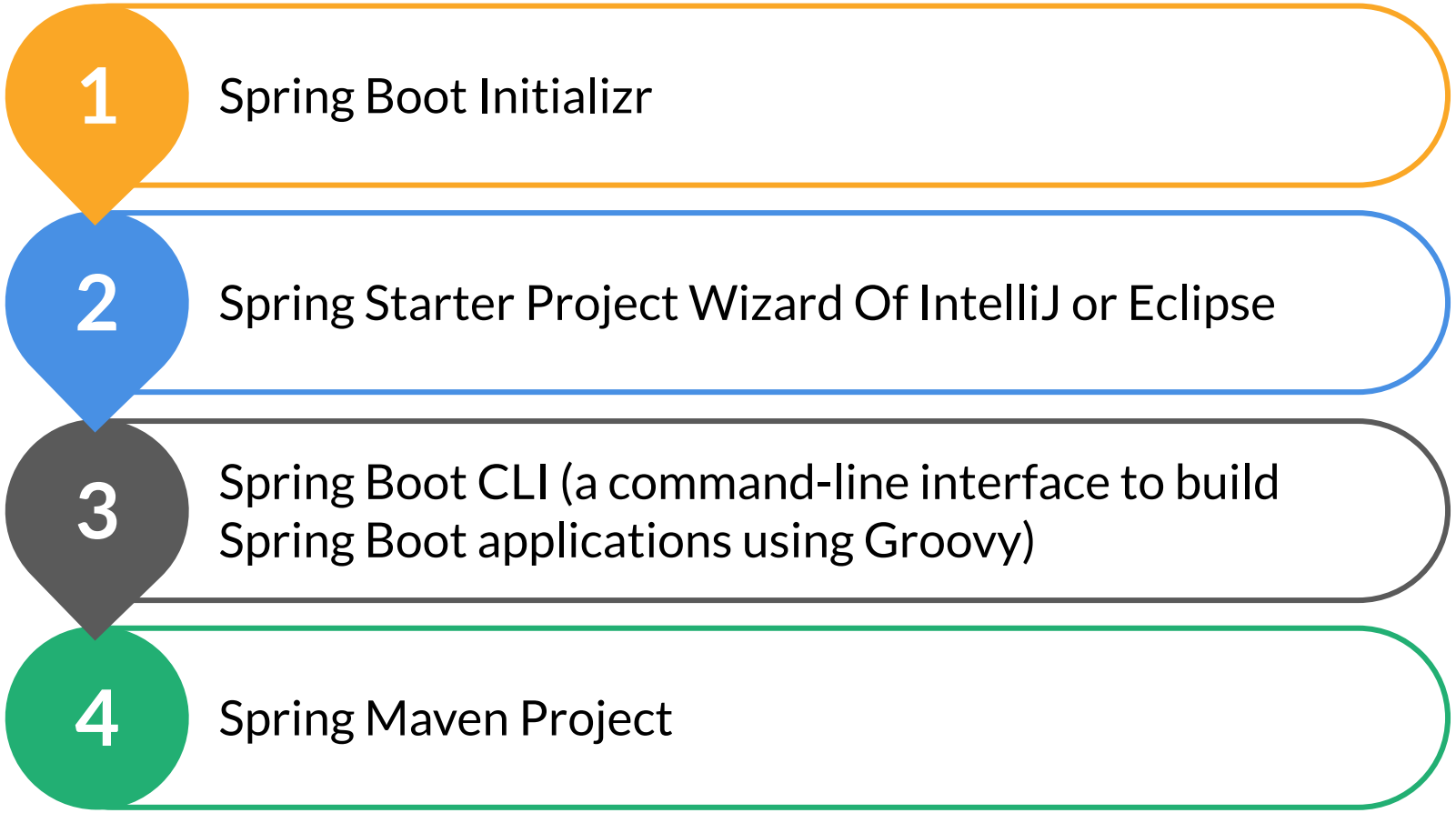


# SPRING VERSUS SPRING BOOT



# Creating a Spring Boot Application

# SPRING BOOT PROJECT CREATION



# SPRING BOOT STARTERS

- Spring Boot Starters are POM files containing a list of all the 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:
  - 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 required to build a RESTful web application using Spring Boot, such as Spring MVC, Tomcat server and hibernate-validator.

# SPRING BOOT STARTERS

- Here 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 building a Spring Boot Project, you do not have to specify the versions of the dependencies or the starter POMs in the pom.xml file of the project.
- You only have to 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 any dependency, then you can do so in the pom.xml file.

# EMBEDDED SERVERS/CONTAINERS

- When building a web application using Java (Servlet/JSP) or Spring, you must perform the following steps:
  - Develop the web application
  - Package it as War
  - Download and install a web server
  - Configure the web server
  - Deploy the application (War packaging) on the server
  - Run the server, which, in turn, runs the application
- Thus, considerable time is spent from Step b to Step f, and this deviates you from your main goal (Step a). So, how can you solve this issue?

# EMBEDDED SERVERS/CONTAINERS

- With Spring Boot, web applications can be packaged as Jar. This Jar 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:
  - Develop the web application
  - Click the run button (just like a normal Java application)
- Spring Boot supports three embedded servers/containers for building applications: Tomcat, Undertow and Jetty.
- Tomcat is the (Opinionated) default embedded server for Spring Boot applications.
- The choice of a container usually depends on factors such as memory requirement, speed, available configuration options, comfort, features and policy.

# SPRING BOOT AUTO-CONFIGURATION

- Spring Boot is nothing but ‘Spring on steroids’ or ‘Spring with auto-configuration’.
- Normally, auto-configuration is triggered by annotating the Main class with **@SpringBootApplication** annotation.
- This annotation is a combination of 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.

# Web Application

# WEB APPLICATION

- What is a Web Application?
  - An application that is installed on a remote machine and hosted using web servers such as **Apache Tomcat** and **Jetty** is known as a web application.
  - It is accessed via a public network called the **Internet** through the **HTTP** protocol.
  - End users need to use a **web browser** to access the web application.
  - Example: Facebook is a web application that is installed on a remote machine and hosted using a server whose address is https://www.facebook.com/, and it can be accessed using a web browser such as Google Chrome.

# WEB APPLICATION: SEGMENTS

- 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.

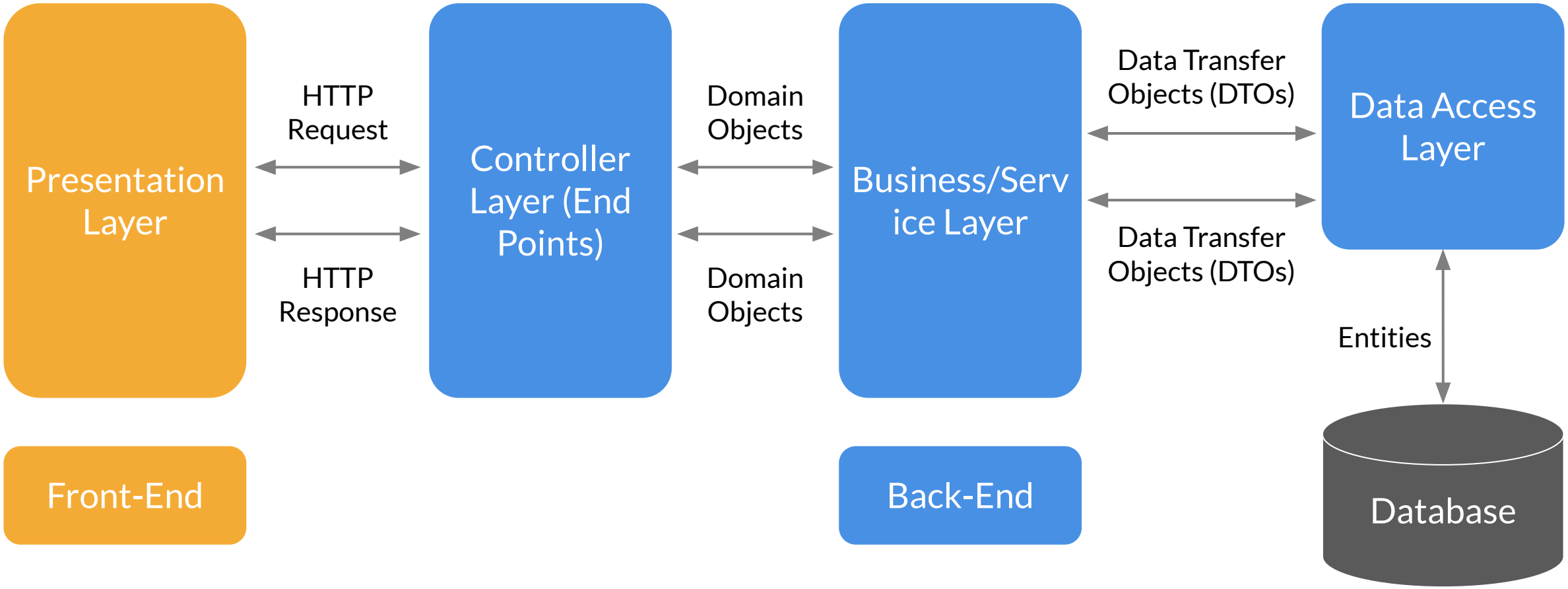
# LAYERED ARCHITECTURE

- You can build the **back end** and the **front end** of a web application **separately**, which will **interact** with each other using **HTTP requests and responses**.
- You also know that you should build applications (back end and front end) in a **loosely coupled** manner so that they are maintainable.
- **Layered Architecture** is one such technique that helps you build loosely coupled applications by breaking down an application into layers and organising its different components into different layers based on the roles and responsibilities carried out by those components.
- All these layers (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).

# LAYERED ARCHITECTURE

- Typically, a web application is distributed into three layers:
  - Data access layer
  - Service/Business layer
  - Presentation/Controller layer
- Each layer has separate roles and contributes differently to the application.

# LAYERED ARCHITECTURE: WEB APPLICATION



# DATA ACCESS LAYER

- This layer is a part of the back-end segment.
- All the components (basically, beans) responsible for interacting with the database will be a part of this layer.
- The application has a package inside that contains all the classes whose objects will be used either for storing data into the database or for fetching data from the database.
- The components inside this layer will not be dependent on the components of the other two layers.

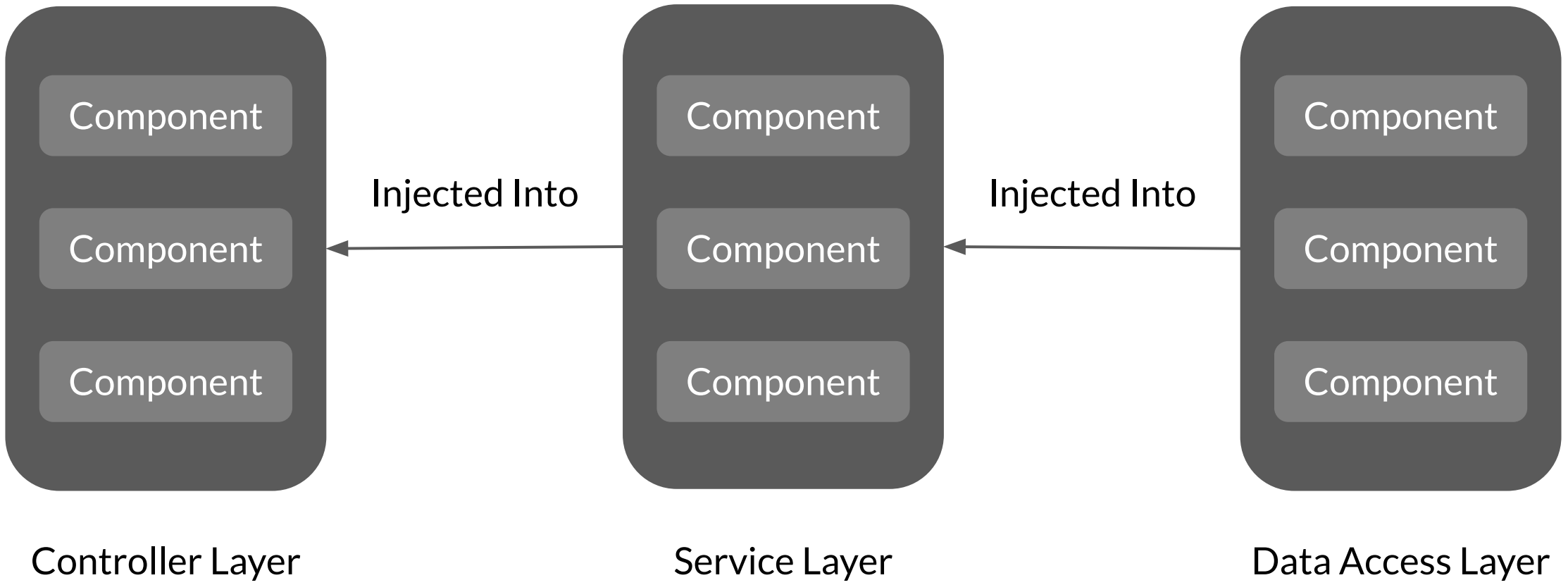
# SERVICE LAYER

- This layer is also a part of the back-end segment.
- It consists of all components that are responsible for processing the client request, based on the business logic, and preparing the response.
- The application has a separate package inside that contains all the classes whose objects will be used for performing the business logic.
- 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

- This layer is a part of 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 application has a separate package inside that contains all the classes whose objects will be used for exposing the end points.
- 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.

# LAYERED ARCHITECTURE: DEPENDENCY FLOW



# LAYERED ARCHITECTURE

Consider an online shopping website. The table here presents the various functions performed by the different layers.

| Front End | Back End | |
|---|---|---|
| Presentation Layer | Service/Business Layer | Data Access Layer |
| Displaying all product images using web page GUI<br><br>Displaying all valid information about products on web page GUI | Selecting and adding products to the cart<br><br>Calculating bills and discounts and generating purchase order invoice | Storing customer details such as billing and orders into the database<br><br>Retrieving product information from the database<br><br>Retrieving customer comments related to products from the database |

# Configuring a Spring Boot Project

# APPLICATION.PROPERTIES CONFIGURATION FILE

- While creating beans, Spring Boot first checks whether the developer has provided custom properties for that bean in the application.properties file.
- If the dependency needs 10 properties to be instantiated as bean, but the developer has provided values for only 2 of the properties, the Spring Boot creates a bean using 2 custom values and 8 default values.
- 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 it by defining the following property in the application.properties file: server.port=8081.

# ALTERNATIVES

- The application.properties file is one of the means for providing custom properties to override the opinionated defaults used by Spring Boot.
- Here are some alternatives based on order of precedence:
  - Command-line arguments,
  - Java system properties,
  - Environment variables
  - YAML files.
- With these, you can externalise your configuration such that the same application code can work in different environments.

# COMMAND-LINE ARGUMENTS

- You can provide custom properties using command-line arguments by prefixing the arguments with '--' (a double hyphen).
- For example, if you want to run an application on server port 8081, you can do so by using command-line arguments as shown below:
- java -jar <name of jar file> --server.port="8081"
- Command-line properties take precedence over any other source of properties.

# JAVA SYSTEM PROPERTIES

- You can also provide custom configurations using Java system properties.
- You can set a Java system property in one of the following two ways:
  - java -Dserver.port="8081" -jar <nameOfJar>
  - System.setProperty("server.port", "8081")
- You can also access Java system properties as shown below:
  - System.getProperty("server.port")

# ENVIRONMENT VARIABLES

- You can provide custom configurations using environment variables.
- You can set environment variables just before running your application as shown below:
  - export SERVER_PORT="8081"

# YAML FILES

- YAML files are exactly the same as properties files and both follow the same rules. The only difference is that YAML files are more convenient when dealing with data of hierarchical configuration.
- For example, say you want to provide a custom configuration for a database. You can do so using application.properties as shown below:
  - spring.datasource.url=jdbc:mysql://localhost:3306/TEST
  - spring.datasource.username=root
  - spring.datasource.password=root
- You can do the same using a YAML file as shown below:
  - spring:
    - datasource:
      - url: jdbc:mysql://localhost:3306/TEST
      - username: root
      - password: root

# YAML FILES

- Due to the absence of repetitive prefixes and the hierarchical configuration, the YAML configuration looks clearer and more readable.
- To use YAML files for a custom configuration, you need to add the SnakeYAML library to the classpath.
- A YAML file is also a better choice for interacting with other frameworks or libraries since properties files can be used only in Java projects.
- You can load YAML files using YamlPropertiesFactoryBean (load YAML as Properties) or YamlMapFactoryBean (load YAML as Map) classes.

# CUSTOM CONFIGURATION USING PROFILES

- An application goes through multiple stages during development, typically ‘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 help you provide environment-specific properties for your applications.

# CUSTOM CONFIGURATION USING PROFILES

- 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 one properties file for the default settings. Here are some examples of such files:
  - application.properties
  - application-dev.properties
  - application-testing.properties
  - application-prod.properties
- You can also control bean creation for specific profiles by marking a component class with the @Profile("profile-name") annotation.

# SET ACTIVE PROFILES

- 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