Spring & SpringBoot

Spring	Spring Boot
Spring is an open-source lightweight framework widely used to develop enterprise applications.	Spring Boot is built on top of the conventional spring framework, widely used to develop REST APIs.
The most important feature of the Spring Framework is dependency injection.	The most important feature of the Spring Boot is Autoconfiguration.
It helps to create a loosely coupled application.	It helps to create a stand-alone application.
To run the Spring application, we need to set the server explicitly.	Spring Boot provides embedded servers such as Tomcat and Jetty etc.
To run the Spring application, a deployment descriptor is required.	There is no requirement for a deployment descriptor.
To create a Spring application, the developers write lots of code.	It reduces the lines of code.
It doesn't provide support for the in-memory database.	It provides support for the in-memory database such as H2.
Developers need to write boilerplate code for smaller tasks.	In Spring Boot, there is reduction in boilerplate code.
Developers have to define dependencies manually in the pom.xml file.	pom.xml file internally handles the required dependencies.

1. Why do we need Spring Boot if Spring already exists?

Answer:
Spring provides a powerful framework but requires a lot of manual configuration (XML,

Java-based, annotations). This slows down development, especially when building multiple small applications like microservices. Spring Boot reduces boilerplate by offering:

- **Autoconfiguration** (detects dependencies and configures them automatically).
- **Starter dependencies** (bundled JARs for common use cases).
- **Embedded servers** (like Tomcat/Jetty, so no need to deploy WAR files).
  This enables developers to build applications much faster and more efficiently.

---

2. What problem does Spring Boot solve compared to Spring?

Answer:
Spring Boot simplifies the setup of Spring applications. With Spring, you must configure DispatcherServlet, ViewResolver, DataSource, etc., manually. Spring Boot **auto-configures** these based on the classpath and existing configurations, so you can focus on business logic instead of infrastructure setup.

---

3. Can you explain Spring Boot Autoconfiguration?

Answer:
Autoconfiguration is a feature where Spring Boot automatically configures beans based on the dependencies present in the classpath.

- Example: If Hibernate and a DataSource are available, Spring Boot will automatically configure JPA and an in-memory database without extra setup.
  This minimizes repetitive configuration.

---

4. What are Spring Boot Starters?

Answer:
Spring Boot Starters are pre-defined dependency descriptors that bundle commonly used libraries into a single unit.

- Example: spring-boot-starter-web includes Tomcat, Spring MVC, and JSON support.
  This saves time since you don’t need to search and add multiple individual dependencies.

---

5. Why is Spring Boot preferred in Microservices architecture?

Answer:
In microservices, we build multiple small services instead of one large application. Each service should be:

- Quick to develop,
- Lightweight,
- Independently deployable.
  Spring Boot helps by:
- Eliminating boilerplate configuration,
- Providing embedded servers (no WAR deployment needed),
- Supporting production-ready features like monitoring, metrics, and health checks.

S.No.	SPRING MVC	SPRING BOOT
1.	Spring MVC is a Model View, and Controller based web framework widely used to develop web applications.	Spring Boot is built on top of the conventional spring framework, widely used to develop REST APIs.
2.	If we are using Spring MVC, we need to build the configuration manually.	If we are using Spring Boot, there is no need to build the configuration manually.
3.	In the Spring MVC, a deployment descriptor is required.	In the Spring Boot, there is no need for a deployment descriptor.
4.	Spring MVC specifies each dependency separately.	It wraps the dependencies together in a single unit.
5.	Spring MVC framework consists of four components : Model, View, Controller, and Front Controller.	There are four main layers in Spring Boot: Presentation Layer, Data Access Layer, Service Layer, and Integration Layer.
6.	It takes more time in development.	It reduces development time and increases productivity.
7.	Spring MVC do not provide powerful batch processing.	Powerful batch processing is provided by Spring Boot.
8.	Ready to use feature are provided by it for building web applications.	Default configurations are provided by it for building a Spring powered framework.

Definition of Bean

- A **Bean** in Spring is an object that is **managed by the Spring IoC container**.
- These are the building blocks of a Spring application.
- Beans are created, configured, and managed by the container based on configuration metadata (XML, Java Config, or Annotations).

👉 **Interview-friendly definition:**
A bean is simply a Java object that is created, initialized, assembled, and managed by the Spring IoC container.

Definition of IoC (Inversion of Control)

- **Inversion of Control (IoC)** is a design principle where the control of object creation and dependency management is shifted from the developer to the Spring container.
- Instead of using new to create objects, Spring IoC container takes care of creating objects (beans), injecting dependencies, and managing their life cycle.
- IoC in Spring is mainly implemented through **Dependency Injection (DI)**.

👉 **Interview-friendly definition:**
IoC is a principle where the responsibility of creating and managing objects is given to the Spring container instead of the developer. In Spring, IoC is implemented using Dependency Injection.

Bean

- A **Bean** is the **object itself** that your application needs.
- Example: Car, Engine, UserService.
- Beans are **created and managed by Spring IoC container**.
- Without Spring, you would write:

Engine engine = new Engine();

Car car = new Car(engine);

With Spring, you define a Bean and let the container manage it:

@Component

```
class Engine {}

@Component
class Car {
  @Autowired
  Engine engine;
}
```

👉 **Think of Bean = the "dish" you ordered in a restaurant.**

🔵 **IoC (Inversion of Control)**

- **IoC is not an object** → it's a **principle/design pattern**.
- It means **you don't create objects yourself**; instead, the **container (Spring) creates and wires them for you**.
- IoC is implemented by **Dependency Injection**.

👉 **Think of IoC = the "restaurant system" that cooks and serves dishes instead of you cooking at home.**

Bean

- **Definition:** A Bean is just an **object created and managed by Spring**.
- **Example:** Think of a **Car** object. In Spring, the Car is a **bean** because the container creates and manages it for you.

🔄 **IoC (Inversion of Control)**

- **Definition:** IoC means the **control of object creation is taken away from you and given to Spring container**.
- **Example:** Normally, you would cook food at home (you create objects). In IoC, you go to a **restaurant** and the **kitchen (container)** prepares the food. You just order it.

🔌 **DI (Dependency Injection)**

- **Definition:** DI is the **way IoC works** — Spring **injects the required objects (dependencies) into a bean** automatically.
- **Example:** A Car needs an Engine. With DI, Spring automatically **injects the Engine into the Car** instead of you manually creating and setting it.

✅ **One-Liner Recap for Interview:**

- **Bean** = the object (Car).
- **IoC** = container controls creation (Restaurant makes the dish).
- **DI** = container injects what is needed (Car gets Engine automatically).

Spring Container

Definition:
The *Spring Container* is the core of the Spring Framework. It is responsible for **creating, configuring, and managing the lifecycle of beans** (objects) in a Spring application.

Think of it as a factory that knows:

1. What objects (beans) are needed.
2. How to create them.
3. How to configure and connect them together.

Key Points:

- It uses **IoC (Inversion of Control)** to manage objects.
- It supports **DI (Dependency Injection)** to automatically provide dependencies.
- Spring provides different types of containers, like:

- **BeanFactory** – Basic container, lightweight, supports lazy-loading.
- **ApplicationContext** – Advanced container, supports internationalization, events, and more.

Three are two Types of Ioc Containers

i)BeanFactory

ii)ApplicationContext

1️⃣ BeanFactory

Definition:
BeanFactory is the **basic Spring container** that provides the fundamental **IoC (Inversion of Control)** functionality. It is responsible for **creating and managing beans** in a Spring application.

Key Points:

- It is **lightweight** and consumes less memory.
- Beans are **created lazily** (i.e., only when requested).
- Basic container, mainly used for **simple scenarios** or memory-sensitive applications.

Analogy:
Think of it like a **vending machine**:

- You press a button (request a bean).
- The machine prepares and gives you the drink (creates the bean **only when needed**).

Example:

BeanFactory factory = new XmlBeanFactory(new FileSystemResource("beans.xml"));

Car car = (Car) factory.getBean("car"); // bean created only now

---

2️⃣ ApplicationContext

Definition:
ApplicationContext is the **advanced Spring container** built on top of BeanFactory. It not only provides **IoC and DI**, but also **additional enterprise features**.

Key Features:

- Beans are **preloaded at startup** (eager initialization).
- Supports:
    - Internationalization (i18n)
    - Event propagation
    - Annotation-based configuration
    - Access to resources like files, URLs, etc.

Analogy:
Think of it like a **restaurant kitchen**:

- Everything is prepared in advance (beans are loaded at startup).
- Ingredients (dependencies) are already placed.
- The chef (container) serves ready-to-use dishes whenever needed.

Example:

ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");

Car car = context.getBean("car", Car.class); // bean already created

BeanFactory → Basic container, lazy beans, like a vending machine.

ApplicationContext → Advanced container, preloaded beans, like a full-service kitchen.

Key Features of IoC Container

The key features of IoC Container are listed below

- **Dependency Injection:** Automatically injects dependencies into our classes.
- **Lifecycle Management:** Manages the lifecycle of beans, including instantiation, initialization and deletion.
- **Configuration Flexibility:** Supports both XML-based and annotation-based configurations.

- **Loose Coupling:** Promotes loose coupling by decoupling the implementation of objects from their usage.

🔌 Dependency Injection (DI) in Spring

Definition:
Dependency Injection (DI) is a design pattern in which objects receive their dependencies from an **external source (Spring container)** rather than creating them internally.

It promotes **loose coupling**, **easier testing**, and **better maintainability**.

In Spring, DI is achieved mainly through **Constructor Injection** or **Setter Injection**.

✅ Need for Dependency Injection

Suppose **Class One** requires an object of **Class Two** to perform its tasks.

- If **Class One** creates new ClassTwo() internally, then **tight coupling** is introduced.
- Tight coupling leads to **difficult maintenance, reduced testability, and fragile systems**.

Solution:
Spring follows **IoC (Inversion of Control)** and provides dependencies via DI.

- Dependencies are injected by the **Spring Container**, based on configuration.
- This achieves **loose coupling** and **code reusability**.

Types of Spring Dependency Injection

Spring provides two primary ways:

1️⃣ Setter Dependency Injection (SDI)

Definition:
Dependencies are injected into a bean **using setter methods** after the bean is created.

When to Use:

- For **optional dependencies**.

- When you want the flexibility to change dependencies after object creation.

```
package com.example;


import org.springframework.beans.factory.annotation.Autowired;
```

Code Example:

```
public class GFG {
   private IGeek geek;
   // Setter method with @Autowired
   @Autowired
   public void setGeek(IGeek geek) {
      this.geek = geek;
   }
}
```

Bean Configuration (XML):

```
<beans xmlns="http://www.springframework.org/schema/beans"
     xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
   <bean id="GFG" class="com.example.GFG">
      <property name="geek" ref="CsvGFG" />
   </bean>
   <bean id="CsvGFG" class="com.example.impl.CsvGFG" />
   <bean id="JsonGFG" class="com.example.impl.JsonGFG" />
</beans>
```

➡️ Here, CsvGFG is injected into GFG using the **setter method**.

2️⃣ Constructor Dependency Injection (CDI)

Definition:
Dependencies are injected into a bean **using constructors** at the time of object creation.

When to Use:

- For **mandatory dependencies**.
- When you want **immutability** (dependencies cannot be changed after creation).

Code Example:

```
package com.example;

public class GFG {

   private IGeek geek;

   // Constructor injection

   public GFG(IGeek geek) {

      this.geek = geek;

   }

}
```

Bean Configuration (XML):

```
<beans xmlns="http://www.springframework.org/schema/beans"

     xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"

     xsi:schemaLocation="

       http://www.springframework.org/schema/beans

       http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

   <bean id="GFG" class="com.example.GFG">

      <constructor-arg>

        <bean class="com.example.impl.CsvGFG" />

      </constructor-arg>

   </bean>
```

```
    <bean id="CsvGFG" class="com.example.impl.CsvGFG" />
    <bean id="JsonGFG" class="com.example.impl.JsonGFG" />
</beans>
```

➡️ Here, CsvGFG is injected into GFG via the **constructor**.

⚖️ Setter DI vs Constructor DI

Feature	Setter DI	Constructor DI
Object Mutability	Mutable – can change dependencies later	Immutable – dependencies fixed at creation
Dependency Requirement	Optional dependencies supported	All dependencies must be provided upfront
Annotation Requirement	Needs `@Autowired` for auto-injection	`@Autowired` optional in single constructor
Testing	Requires framework/manual setters	Easy testing with mock dependencies
Error Handling	May result in partially initialized beans	Fails fast if dependency not provided

---

🌱 Bean Life Cycle in Spring

🔹 Definition

The **Spring Bean Life Cycle** defines the series of steps a bean goes through from **creation to destruction**, managed by the **Spring Container**.

Bean Life Cycle Phases

The lifecycle of a Spring bean consists of the following phases, which are listed below

- **Container Started:** The Spring IoC container is initialized.
- **Bean Instantiated:** The container creates an instance of the bean.
- **Dependencies Injected:** The container injects the dependencies into the bean.
- **Custom init() method:** If the bean implements InitializingBean or has a custom initialization method specified via @PostConstruct or init-method.
- **Bean is Ready:** The bean is now fully initialized and ready to be used.

- **Custom utility method:** This could be any custom method you have defined in your bean.
- **Custom destroy() method:** If the bean implements DisposableBean or has a custom destruction method specified via @PreDestroy or destroy-method, it is called when the container is shutting down.

Bean Scopes in Spring

Definition:
Bean Scopes define **how many objects (instances)** of a bean are created by the **Spring Container** and **how long they live**.

🔹 **Commonly Used Scopes**

1️⃣ **Singleton Scope (Default)**

- **Meaning:** Only **one object (instance)** of the bean is created per Spring IoC container.
- All requests for that bean return the **same shared object**.
- Default scope if no scope is mentioned.

Example:

```
@Component
@Scope("singleton") // or no scope (default)
public class MyBean {}
```

.xml file

```
ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");
MyBean obj1 = context.getBean(MyBean.class);
MyBean obj2 = context.getBean(MyBean.class);
System.out.println(obj1 == obj2); // true (same object)
```

Analogy: Like a **government ID card** – only one unique per person.

2️⃣ **Prototype Scope**

- **Meaning:** A **new object (instance)** is created **every time** the bean is requested.
- Useful when you need separate instances for different users or tasks.

Example:

```
@Component
@Scope("prototype")
public class MyBean {}
```

.xml file

```
ApplicationContext context = new
ClassPathXmlApplicationContext("beans.xml");
MyBean obj1 = context.getBean(MyBean.class);
MyBean obj2 = context.getBean(MyBean.class);
System.out.println(obj1 == obj2); // false (different objects)
```

Difference Between Singleton and Prototype Bean

Singleton	Prototype
Only one instance is created for a single bean definition per Spring IoC container.	A new instance is created for a single bean definition every time a request is made for that bean.
Same object is shared for each request made for that bean. i.e. The same object is returned each time it is injected.	For each new request a new instance is created. i.e. A new object is created each time it is injected.
By default, scope of a bean is singleton. So we don't need to declare a bean as singleton explicitly.	By default, scope is not prototype, so you have to declare the scope of a bean as prototype explicitly.
Singleton scope should be used for stateless beans.	While prototype scope is used for all beans that are stateful.

🔹 **Web-Only Scopes (valid in a web-aware Spring ApplicationContext)**

3️⃣ **Request**

- A **new bean instance** is created for **every HTTP request**.
- Destroyed once the request is completed.

👉 Example: Like a **train ticket** – new ticket for each journey.

4️⃣ **Session**

- One bean instance is created per **HTTP session**.

- Same bean shared across multiple requests from the same user session.

👉 Example: Like a **shopping cart** – persists while the user session is active.

5️⃣ **Global-Session**

- Bean is scoped to a **global HTTP session** (used in portlet-based apps).
- Shared across multiple portlets in the same global session.

👉 Example: Like a **passport** – used globally across multiple systems.

3 Ways to Create a Spring Bean

Spring Beans are the backbone objects of a Spring application, managed by the IoC container. There are **three main approaches** to create them:

---

1. XML Configuration (beans.xml)

- **How it works**: Beans are declared inside an external XML file using <bean> tags.
- **Steps**:
    1. Create a class (e.g., Student).
    2. Define the bean in beans.xml:
    3. <bean id="studentAmiya" class="Student"/>
    4. Load the XML into the Spring container to instantiate the bean.
- **Advantages**: Clear separation of configuration from code.
- **Disadvantages**: Verbose and harder to maintain for large applications.

---

2. Annotation-Based with @Component

- **How it works**: Annotate your class with @Component, and Spring will auto-detect it via **component scanning**.
- **Steps**:
    1. Add @Component("beanName") on the class:

```
@Component("collegeBean")
public class College {
    public void test() {
        System.out.println("Test College Method");
    }
}
```

8. Enable component scanning with @ComponentScan in a configuration class.

- **Advantages**: No need to define beans in XML, more concise and modern.
- **Disadvantages**: Less explicit — scanning can register unintended beans if not carefully managed.

---

3. Java-Based with @Bean Annotation

- **How it works**: Inside a @Configuration class, define methods annotated with @Bean. Each method returns an object that Spring manages as a bean.
- **Steps**:

```
@Configuration
public class CollegeConfig {
    @Bean
    public College collegeBean() {
        return new College();
    }
}
```

9. Spring registers the returned object (College) as a bean with the method name (collegeBean) as its ID.

- **Advantages**: Full control over bean creation and initialization, especially when dealing with third-party classes.
- **Disadvantages**: Slightly more verbose than annotations like @Component.

Comparison

Method	Description	Use Case
XML (<bean>)	Define beans in external XML file	Legacy apps, external config separation
@Component	Annotate classes for auto-detection	Quick setup, when working with Spring Boot or component scanning
@Bean	Define beans in Java @Configuration class	Precise control, third-party libraries, or when fine-grained customization is required

Spring – Autowiring

Autowiring in Spring allows the container to **automatically inject dependencies** into a bean, reducing the need for explicit configuration. It mainly works through the @Autowired annotation and uses **constructor** or **setter injection** internally.

Autowiring Modes

Mode	Description	Example
No (default)	No autowiring is performed. Dependencies must be set manually in XML or code.	<bean id="city" class="sample.City"/>
byName	Matches a property name with a bean ID of the same name. Uses setter injection.	<bean id="city" class="sample.City" autowire="byName"/>
byType	Injects dependency based on matching **type** (class). Fails if more than one matching type exists.	<bean id="city" class="sample.City" autowire="byType"/>
constructor	Similar to byType, but applies to **constructor arguments**. Fails if multiple candidates exist.	<bean id="city" class="sample.City" autowire="constructor"/>

Mode	Description	Example
autodetect	(Deprecated since Spring 3.0) First tries constructor, then falls back to byType.	<bean id="city" class="sample.City" autowire="autodetect"/>

✅ Advantages

- Less boilerplate code (no explicit setter/constructor injection in config).
- Cleaner configuration and reduced XML.

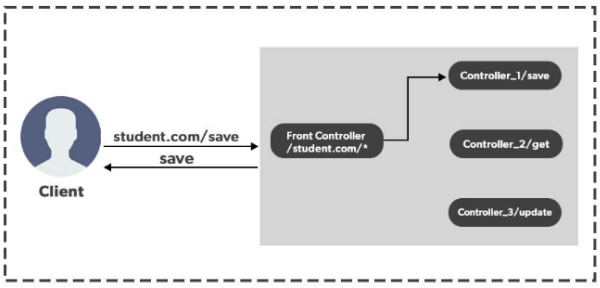
❌ Disadvantages

- Less control for the developer (Spring decides injection strategy).
- Cannot inject **primitive** or **String** values.
- Ambiguity when multiple beans of the same type exist.

What is Dispatcher Servlet in Spring?

Last Updated : 07 Aug, 2025

DispatcherServlet is the Front Controller in a Spring web application. It acts as the entry point for all incoming HTTP requests. When a user makes a request (e.g., student.com/save), the DispatcherServlet receives it first, then decides which controller should handle it (e.g., Controller_1 for /save). After the controller processes the request, the response is sent back to the user.



DispatcherServlet handles incoming HTTP requests and delegates them to the appropriate components. It works with HandlerAdapter interfaces configured in the Spring application to process the request. It uses annotations like @Controller, @RequestMapping, etc., to identify handler methods (controller endpoints) and prepares the appropriate response objects based on the controller's output.

🔹 What is Maven?

Maven is a **build automation and dependency management tool** for Java projects.
It:

- Compiles code.
- Manages dependencies.
- Runs tests.
- Packages the application (JAR/WAR).
- Can deploy artifacts to repositories.

🔹 Spring Boot + Maven

Spring Boot integrates smoothly with Maven. Typically:

1. A pom.xml defines project metadata and dependencies.
2. Spring Boot provides a **Maven plugin** to build executable JARs/WARs that include an embedded Tomcat/Jetty/Undertow server.

Spring-Boot

Spring Boot - Architecture

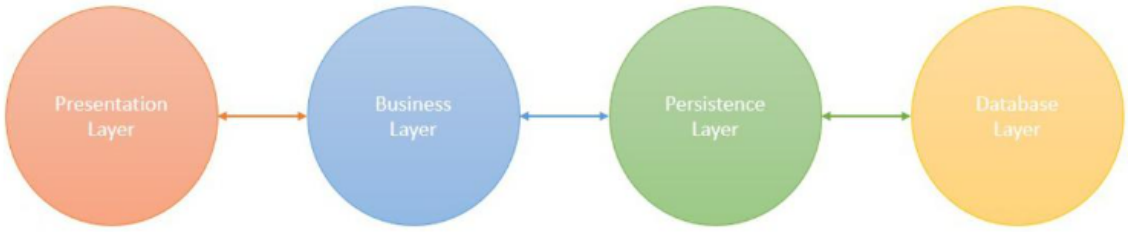
Last Updated : 21 Aug, 2025

Spring Boot is built on top of the Spring Framework and follows a layered architecture. Its primary goal is to simplify application development by providing auto-configuration, embedded servers and a production-ready environment out of the box.

The architecture of Spring Boot can be divided into several layers and components, each playing an important role in building and running modern applications.

Spring Boot Architecture Layers

Spring Boot consists of the following four layers:



1. Presentation Layer

- Handles HTTP requests through REST controllers (GET, POST, PUT, DELETE).
- Manages authentication, request validation and JSON serialization/deserialization.
- Forwards processed requests to the Business Layer for further logic.

2. Business Layer

The Business Layer is responsible for implementing the application's core logic. It consists of service classes that:

- Process and validate data.
- Handle authentication and authorization (integrating Spring Security if needed).
- Apply transaction management using @Transactional.
- Interact with the Persistence Layer to store or retrieve data.

3. Persistence Layer

The Persistence Layer manages database transactions and storage logic. It consists of repository classes using Spring Data JPA, Hibernate or R2DBC for data access. It is responsible for:

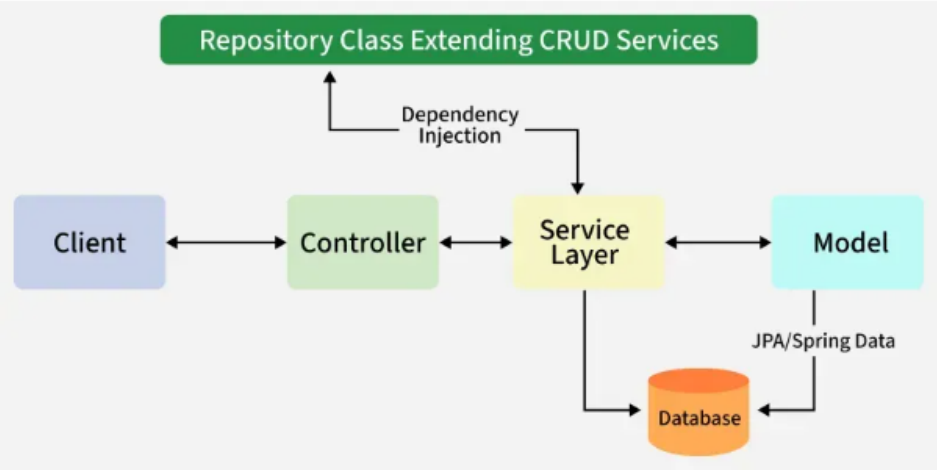
- Mapping Java objects to database records using ORM frameworks.
- Managing CRUD (Create, Read, Update, Delete) operations.
- Supporting relational and NoSQL databases.

4. Database Layer

The Database Layer contains the actual database where the application data is stored. It can support:

- Relational Databases (MySQL, PostgreSQL, Oracle, SQL Server).
- NoSQL Databases (MongoDB, Cassandra, DynamoDB, Firebase).
- Cloud-based databases for scalability.

Spring Boot Flow Architecture



Architecture flow

Explanation:

- The client (frontend or API consumer) sends an HTTP request (GET, POST, PUT, DELETE) to the application.
- The request is handled by the Controller Layer, which maps the request to a specific handler method.
- The Service Layer processes business logic and communicates with the Persistence Layer to fetch or modify data.
- The Persistence Layer interacts with the Database Layer using Spring Data JPA or R2DBC, often through a **Repository Class** that extends CRUD services.
- The processed response is returned as JSON.
- Spring Boot Actuator can be used for monitoring and health checks.

@SpringBootApplication

@SpringBootApplication is a **meta-annotation** in Spring Boot, used on the **main class** to bootstrap a Spring Boot application.

It combines three annotations:

- **@SpringBootConfiguration** → Equivalent to @Configuration. Marks the class as a source of bean definitions.
- **@EnableAutoConfiguration** → Enables Spring Boot's auto-configuration mechanism.
- **@ComponentScan** → Enables scanning of components (@Controller, @Service, @Repository) in the package where the main class resides and its sub-packages.

🔥 **1. @SpringBootConfiguration**

- **Definition**: A specialized form of @Configuration.
- **Purpose**: Marks the class as a **source of bean definitions**.
- **Behind the scenes**:
  - @SpringBootConfiguration itself is annotated with @Configuration.
  - @Configuration tells Spring: "this class contains methods annotated with @Bean, which produce Spring-managed beans."

✅ Example:

```
@SpringBootConfiguration
public class AppConfig {

  @Bean
  public String appName() {
    return "Spring Boot Demo";
  }
}
```

Now, appName() will be registered as a **Spring bean** and can be injected anywhere.

📌 **Interview Point**:

- Difference between @Configuration and @SpringBootConfiguration?
  👉 Functionally, they are the same.
  👉 @SpringBootConfiguration is just a **marker for Spring Boot apps** (so Boot tools can identify the main configuration class).

🔥 **2. @EnableAutoConfiguration**

- **Definition**: Tells Spring Boot to **auto-configure beans** based on the classpath dependencies, property settings, and beans already defined by the user.
- **Mechanism**:
    1. Looks into META-INF/spring.factories files provided by Spring Boot JARs.
    2. Finds all @Configuration classes listed under EnableAutoConfiguration.
    3. Loads them conditionally using @ConditionalOnClass, @ConditionalOnMissingBean, etc.

✅ Example:

- If you add **spring-boot-starter-web** dependency:
    - DispatcherServlet (Spring MVC's front controller) is auto-configured.
    - An **embedded Tomcat** server is started automatically.
    - ObjectMapper for JSON serialization is created.

📌 **Interview Point**:

- What if you don't want a particular auto-configuration?
- @SpringBootApplication(exclude = { DataSourceAutoConfiguration.class })
- public class DemoApplication {}

👉 This prevents Spring Boot from auto-configuring a DataSource.

🔥 **3. @ComponentScan**

- **Definition**: Tells Spring where to look for **Spring-managed components**.
- **What it scans for**:
    - @Component
    - @Controller
    - @Service
    - @Repository
    - @Configuration

- **Default behavior**: Scans the **package of the class** where @SpringBootApplication is located and all its **sub-packages**.

✅ Example:

com.example.demo → @SpringBootApplication here

├── controller → scanned automatically

├── service → scanned automatically

└── repository → scanned automatically

But if your beans are outside the base package, you must specify:

```
@SpringBootApplication(scanBasePackages = {"com.example.demo", "com.other"})
public class DemoApplication {}
```

📌 **Interview Point**:

- Why do we place the @SpringBootApplication class at the **root package**?
  👉 So that @ComponentScan can automatically cover all sub-packages. If placed incorrectly, beans may not be detected.

⚡ **Putting It All Together**

```
@SpringBootApplication  // = @SpringBootConfiguration + @EnableAutoConfiguration + @ComponentScan
public class DemoApplication {
  public static void main(String[] args) {
    SpringApplication.run(DemoApplication.class, args);
  }
}
```

🌐 Spring MVC/Web Annotations Explained

---

1. @Controller

👉 Marks a class as a **Spring MVC Controller** (the "C" in MVC).

- Handles HTTP requests and decides which **view (JSP/Thymeleaf/HTML)** should be returned.
- Works with Model to pass data to the view.

📌 **Analogy**: Think of it as a **traffic officer** — it receives a request, prepares info, and sends you to the correct web page.

Example:

```
@Controller
public class HomeController {

  @GetMapping("/welcome")
  public String welcome(Model model) {
    model.addAttribute("message", "Hello, Spring MVC!");
    return "welcome"; // View name (welcome.jsp / welcome.html)
  }
}
```

➞ URL: /welcome → returns **HTML page** with the message.

---

2. @RestController

👉 Specialized version of @Controller.

- **@Controller + @ResponseBody combined**.
- Instead of returning a view, it directly returns **data (JSON/XML)**.
- Used in **REST APIs**.

📌 **Analogy**: Think of it as a **data server** — instead of sending you to a page, it gives you raw data.

Example:

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

  @GetMapping("/user")
  public User getUser() {
    return new User(1, "Amit"); // Auto converted to JSON
  }
}
```

➡ URL: /api/user → returns JSON:

```
{ "id": 1, "name": "Amit" }
```

---

3. @RequestMapping

👉 Maps a **URL request** to a controller class/method.

- Can be used at **class level** (base URL) and **method level** (endpoint).
- Can restrict by HTTP method (GET, POST, etc.).

📌 **Analogy**: Think of it as a **GPS navigation rule** — it tells Spring, *"when this URL is called, go to this method."*

Example:

```
@Controller
@RequestMapping("/products")
public class ProductController {

  @RequestMapping(value="/list", method=RequestMethod.GET)
  public String showProducts(Model model) {
```

```
    model.addAttribute("products", List.of("Laptop", "Phone"));
    return "productList";
  }
}
```

➞ URL: /products/list → returns a product list page.

---

4. @RequestParam

👉 Extracts **query parameters / form parameters** from the request and binds them to method arguments.

📌 **Analogy**: Think of it as **reading search box input** from a URL.

Example:

```
@RestController
public class SearchController {

  @GetMapping("/search")
  public String search(@RequestParam("q") String query) {
    return "You searched for: " + query;
  }
}
```

➞ URL: /search?q=laptop
➞ Response: "You searched for: laptop"

---

5. @PathVariable

👉 Extracts values from the **URL path itself**.

- Used when parameters are part of the path instead of query string.

📌 **Analogy**: Think of it as **reading the name written on the door** instead of asking.

Example:

```
@RestController
```

```
@RequestMapping("/users")
public class UserController {

  @GetMapping("/{id}")
  public String getUser(@PathVariable("id") int userId) {
    return "User ID: " + userId;
  }
}
```

➞ URL: /users/101
➞ Response: "User ID: 101"

---

6. @RequestBody

👉 Maps the **HTTP request body** (JSON/XML) into a Java object.

- Used in POST/PUT requests when client sends JSON.

📌 **Analogy**: Think of it as **unpacking a parcel** into a Java object.

Example:

```
@RestController
@RequestMapping("/api/orders")
public class OrderController {

  @PostMapping
  public String placeOrder(@RequestBody Order order) {
    return "Order placed for " + order.getProductName();
  }
}
```

➞ Request: POST /api/orders
Body: { "productName": "Laptop", "quantity": 2 }
➞ Response: "Order placed for Laptop"

---

7. @ResponseBody

👉 Tells Spring to write the **return value of a method** directly into the HTTP response (instead of resolving a view).

- Common with @Controller when returning raw data.
- Built into @RestController (so you don't need it there).

📌 **Analogy**: Think of it as **talking directly** instead of giving someone a note (view).

Example:

```
@Controller
public class GreetingController {

  @GetMapping("/greet")
  @ResponseBody
  public String greet() {
    return "Hello, Spring!";
  }
}
```

➞ URL: /greet
➞ Response: "Hello, Spring!"

---

8. @ModelAttribute

👉 Used for **binding form data** to a model object OR for adding attributes to the model before a request.

📌 **Analogy**: Think of it as **auto-filling a form** into a Java object.

Example 1 – Binding form data

```
@Controller
public class UserController {
```

```
  @PostMapping("/register")
  public String register(@ModelAttribute User user) {
    System.out.println("User Name: " + user.getName());
    return "success";
  }
}
```

➞ If form fields are name=email, Spring will populate User object automatically.

Example 2 – Pre-populating model data

```
@Controller
public class CommonController {

  @ModelAttribute("countries")
  public List<String> populateCountries() {
    return List.of("India", "USA", "UK");
  }

  @GetMapping("/signup")
  public String signupForm() {
    return "signup"; // "countries" available in signup.jsp
  }
}
```

🔑 Spring Boot Starters

- **What they are**:
  - Pre-defined **dependency descriptors** that group commonly used libraries together.

    - Added in pom.xml (Maven) or build.gradle (Gradle).
- **Why they exist**:
    - Before Spring Boot → Devs had to manually add 5–10 dependencies (with versions).
    - With Starters → One line gives you all required dependencies with **compatible versions**.

✅ **Examples**

- spring-boot-starter-web → Spring MVC + Jackson + Embedded Tomcat
- spring-boot-starter-data-jpa → Spring Data JPA + Hibernate + JDBC
- spring-boot-starter-security → Spring Security
- spring-boot-starter-test → JUnit + Mockito + Spring Test

🚀 **Advantages**

1. **Boosts productivity** → less time spent managing dependencies.
2. **Simplifies POM** → fewer entries in pom.xml.
3. **Production-ready** → dependencies are tested to work together.
4. **Version management handled** → uses **Spring Boot BOM** (Bill of Materials) to align versions.

🎯 **Interview Takeaway**

👉 **Spring Boot Starters** = One-stop dependencies for specific features, removing manual dependency management and ensuring version compatibility.

🔑 Spring Boot – application.properties / application.yml

- **Definition**:
  A configuration file in Spring Boot used to **externalize application settings** instead of hardcoding them in code.
- **Use**:
  - Set **server settings** (e.g., server.port=8081)
  - Configure **databases** (URL, username, password)
  - Control **logging levels**
  - Manage **environment-specific profiles** (dev, test, prod)

👉 In short: application.properties / application.yml is used to **customize and manage application behavior** in a flexible, environment-specific way.

🔑 Spring Boot Actuator

Definition

Spring Boot Actuator is a set of **built-in production-ready features** that help you monitor and manage your application.

It exposes operational information (health, metrics, logs, env, etc.) via **REST endpoints** and/or **JMX**.

Use

- Monitor **application health** (/actuator/health)
- View runtime **metrics** (memory, threads, CPU, DB connections, HTTP requests)
- Access **application info** (/actuator/info)
- Manage **log levels** at runtime
- Integrate with monitoring tools (Prometheus, Grafana, etc.)

🔑 Spring Boot - Logging

Definition

Logging in Spring Boot is the mechanism of **recording runtime information, actions, errors, and system events**.
It helps in **monitoring application performance**, **debugging issues**, and **auditing application behavior**.

Why use Logging in Spring Boot?

- Understand what's happening inside the application.
- Diagnose unusual behavior or errors.
- Monitor application performance.
- Maintain an audit trail.

Elements of a Logging Framework

1. **Logger** → Captures log messages.
2. **Formatter** → Formats log messages (timestamp, log level, etc.).
3. **Handler/Appender** → Outputs logs (console, file, email, monitoring system).

Spring Boot – RESTful Web Services

REST (**Representational State Transfer**) is an **architectural style** for building web services that use standard **HTTP methods (GET, POST, PUT, DELETE)** to perform operations on **resources** identified by **URIs**.
It is **stateless**, **lightweight**, and typically exchanges data in **JSON** format.

🔑 Serializable & Deserializable in Spring Boot

Definition

1. **Serializable**:

- The process of converting a **Java object** into a **byte stream** so that it can be **stored** (file, database) or **transmitted** (network).

- In Spring Boot, this is commonly used when sending objects over **HTTP (JSON)** or storing in sessions.

2. **Deserializable**:

- The process of converting a **byte stream** (or JSON/XML) back into a **Java object**.
- In Spring Boot, this happens automatically when receiving **JSON in a REST API** via @RequestBody.

Core Concept in Spring Boot

- When a client sends JSON to a Spring Boot REST API:
    - Spring **deserializes** JSON into a Java object.
- When Spring returns a Java object in a response:
    - Spring **serializes** it into JSON automatically.

This uses **Jackson library** by default in Spring Boot.

1. Create a Java class (Model)

```
package com.example.demo.model;
import java.io.Serializable;
public class User implements Serializable {
  private int id;
  private String name;
  // Default constructor
  public User() {}
  // Constructor
  public User(int id, String name) {
    this.id = id;
    this.name = name;
```

```
    }
    // Getters and Setters
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
}
```

2. Create a REST Controller

```
package com.example.demo.controller;
import com.example.demo.model.User;
import org.springframework.web.bind.annotation.*;
@RestController
@RequestMapping("/users")
public class UserController {
    // Serialize Java object to JSON
    @GetMapping("/{id}")
    public User getUser(@PathVariable int id) {
        return new User(id, "Amit"); // Spring Boot serializes this to JSON
    }
    // Deserialize JSON to Java object
    @PostMapping
    public String createUser(@RequestBody User user) {
        return "User created: " + user.getName(); // JSON from client is deserialized into
User object
    }
```

}

1. **GET Request**

bash Copy code

```
GET /users/1
```

Response (JSON):

json Copy code

```
{
  "id": 1,
  "name": "Amit"
}
```

- Java object `User` → serialized to JSON.

2. **POST Request**

bash Copy code

```
POST /users
Content-Type: application/json

{
  "id": 2,
  "name": "Rahul"
}
```

Response:

sql Copy code

```
User created: Rahul
```

- JSON → deserialized into Java `User` object.

Exception Handling in Spring Boot

✅ Why Exception Handling is Used?

- In real-world apps, errors are inevitable (e.g., invalid input, DB failure, null values).
- **Exception Handling** ensures:
    1. The application doesn't crash abruptly.

2. Users get **meaningful error responses** instead of raw stack traces.
3. Developers can maintain logs for debugging.

1. Checked vs Unchecked Exceptions

✅ **Checked Exceptions**

- Inherit from Exception.
- Must be **handled** at compile time (try/catch or throws).
- Represent **expected problems**.
- Example: IOException, SQLException.

```
try {
   FileReader file = new FileReader("test.txt");
} catch (IOException e) {
   System.out.println("File not found!");
}
```

👉 Compile-time error if not handled.

✅ **Unchecked Exceptions**

- Inherit from RuntimeException.
- Not forced to handle at compile time.
- Represent **programming errors**.
- Example: NullPointerException, ArrayIndexOutOfBoundsException.

```
String name = null;
System.out.println(name.length()); // NullPointerException
```

👉 App crashes if not handled.

🔥 **2. Normal Exception Handling (@ExceptionHandler)**

Used inside one controller to catch specific exceptions.

@RestController

```
@RequestMapping("/users")
public class UserController {

  @GetMapping("/{id}")
  public String getUser(@PathVariable int id) {
    if (id <= 0) {
      throw new IllegalArgumentException("ID must be greater than 0");
    }
    return "User ID: " + id;
  }

  @ExceptionHandler(IllegalArgumentException.class)
  public String handleIllegalArgument(IllegalArgumentException ex) {
    return "Local Error: " + ex.getMessage();
  }
}
```

👉 Input: GET /users/-1
👉 Output:

Local Error: ID must be greater than 0

🔥 3. Global Exception Handling (@ControllerAdvice)

Used for all controllers in the app.

```
@RestControllerAdvice
public class GlobalExceptionHandler {

  @ExceptionHandler(IllegalArgumentException.class)
  public ResponseEntity<String> handleIllegalArgument(IllegalArgumentException ex) {
```

```
    return new ResponseEntity<>("Global Error: " + ex.getMessage(),
HttpStatus.BAD_REQUEST);
  }

  @ExceptionHandler(Exception.class)
  public ResponseEntity<String> handleException(Exception ex) {
    return new ResponseEntity<>("Something went wrong: " + ex.getMessage(),
                    HttpStatus.INTERNAL_SERVER_ERROR);
  }
}
```

👉 Input: GET /users/-1
👉 Output:

```
Global Error: ID must be greater than 0
```

👉 Input: GET /users/abc
👉 Output:

```
Something went wrong: Failed to convert value of type 'java.lang.String' to required
type 'int'
```

🔥 4. Custom Exception Handling

Define business-specific exception classes.

```
public class UserNotFoundException extends RuntimeException {
  public UserNotFoundException(String message) {
    super(message);
  }
}
```

Use in controller:

```
@GetMapping("/{id}")
public String getUser(@PathVariable int id) {
  if (id != 1) {
```

```
        throw new UserNotFoundException("User with ID " + id + " not found");
    }
    return "User ID: " + id;
}
```

Handle globally:

```
@ExceptionHandler(UserNotFoundException.class)
public ResponseEntity<String> handleUserNotFound(UserNotFoundException ex) {
    return new ResponseEntity<>(ex.getMessage(), HttpStatus.NOT_FOUND);
}
```

👉 **Input: GET /users/2**
👉 **Output:**

User with ID 2 not found

🔥 **5. How It Works in Spring Boot (Behind the Scenes)**

- **When an exception occurs, Spring looks for a matching handler:**
    1. **@ExceptionHandler inside the same controller.**
    2. **@ControllerAdvice global handler.**
    3. **If none found → Default Spring error response (whitelabel error page or JSON).**
- **For REST APIs, Spring Boot uses @RestControllerAdvice to return JSON error responses automatically.**

Spring Boot – Validation using Hibernate Validator

✅ **What is Validation?**

Validation ensures that **input data is correct and safe** before being processed or saved.
Example:

- Username should not be empty.
- Email must be valid.
- Age should be greater than 18.

✅ **Why Hibernate Validator?**

- **Hibernate Validator** is the **reference implementation of the Bean Validation API (JSR 380)**.
- Spring Boot integrates it seamlessly for validating request bodies, query params, and path variables.

Common Hibernate Validator Annotations

Hibernate validators provide the following annotations that are very helpful for software development.

1. **@NotNull:** @NotNull ensures that a field is not null but allows empty values (e.g., an empty string or an empty collection).
2. **@NotEmpty:** @NotEmpty ensures that a field is not null and also not empty, meaning it must contain at least one element (for collections) or at least one character (for strings).
3. **@NotBlank:** @NotBlank applies only to strings and ensures they are not null, not empty and contain at least one non-whitespace character (i.e., spaces alone are not allowed).
4. **@Min:** Given Minimum value has to be satisfied
5. **@Max:** Given Maximum value has to be satisfied
6. **@Size:** Field size should be less than or greater than the specified field size
7. **@Email:** Email can be validated with this
8. **@Pattern:** Given the RegEx Pattern has to be satisfied.

Step 1: Create an Entity Class

public class GeekEmployee {

```
@NotBlank(message = "Employee name cannot be blank")
private String geekEmployeeName;
@Email(message = "Email should be valid")
private String geekEmailId;
@Min(value = 10000, message = "Salary must be at least 10,000")
private double salary;
@NotEmpty(message = "Qualifications cannot be empty")
private List<String> qualifications;
// Getters and Setters
public String getGeekEmployeeName() {
   return geekEmployeeName;
}
public void setGeekEmployeeName(String geekEmployeeName) {
   this.geekEmployeeName = geekEmployeeName;
}
public String getGeekEmailId() {
   return geekEmailId;
}
public void setGeekEmailId(String geekEmailId) {
   this.geekEmailId = geekEmailId;
}
public double getSalary() {
   return salary;
}
public void setSalary(double salary) {
   this.salary = salary;
}
```

```
    public List<String> getQualifications() {
        return qualifications;
    }
    public void setQualifications(List<String> qualifications) {
        this.qualifications = qualifications;
    }
}
```

Step 2: Exception handling for Validators Errors

```
@ControllerAdvice
public class ExceptionHandler {
    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<Map<String, String>>
handleValidationExceptions(MethodArgumentNotValidException ex) {
        Map<String, String> errors = new HashMap<>();
        ex.getBindingResult().getFieldErrors().forEach(error ->
                errors.put(error.getField(), error.getDefaultMessage()));
        return new ResponseEntity<>(errors, HttpStatus.BAD_REQUEST);
    }
}
```

Step 3: REST Controller for Validation

```
@RestController
@RequestMapping("/geek")
public class GeekEmployeeController {
    @PostMapping("/addEmployee")
    public ResponseEntity<String> addEmployee(@Valid @RequestBody GeekEmployee
employee) {
```

```
        return new ResponseEntity<>(
                "Employee details are valid and added successfully! \n" +
                "Name: " + employee.getGeekEmployeeName() + "\n" +
                "Email: " + employee.getGeekEmailId() + "\n" +
                "Salary: " + employee.getSalary() + "\n" +
                "Qualifications: " + employee.getQualifications(),
                HttpStatus.OK
        );
    }
}
```

Spring Boot with Database and Data JPA

🔥 1. ORM (Object Relational Mapping)

✅ Definition

ORM is a programming technique that allows developers to map Java objects to relational database tables automatically.

- **Each Java class → Table**
- **Each field → Column**
- **Each object → Row**

✅ Working Example (Conceptual)

```
@Entity
public class User {
  @Id
```

private int id;

private String name;

private String email;

}

➞ ORM will automatically create a table like:

id name email

1 John john@gmail.com

👉 You don't have to write INSERT INTO or SELECT * manually, ORM handles it.

🔥 2. JPA (Java Persistence API)

✅ Definition

- **JPA is a specification (set of rules/standards) for ORM in Java.**
- **It tells what should be done, but doesn't provide actual code.**
- **Think of it as an interface.**

✅ Key Features of JPA

- **Annotations like @Entity, @Table, @Id.**
- **EntityManager for CRUD operations.**
- **Queries using JPQL (Java Persistence Query Language).**

✅ Working Flow

1. **You define entities using JPA annotations.**

2. **You use EntityManager (provided by implementation) to persist/fetch data.**
3. **JPA itself does not do the work → it delegates to a JPA Provider (like Hibernate).**

🔥 3. Hibernate

✅ Definition

- **Hibernate is the most popular JPA implementation (provider).**
- **It provides the actual code behind JPA specifications.**
- **Hibernate can also be used directly (without JPA).**

✅ Features of Hibernate

- **Automatic table creation (DDL).**
- **HQL (Hibernate Query Language).**
- **Caching support.**
- **Lazy loading (load related entities only when needed).**

✅ Example

```
User user = new User(1, "John", "john@gmail.com");

entityManager.persist(user);  // JPA method


// Under the hood, Hibernate converts it into:

INSERT INTO user (id, name, email) VALUES (1, 'John', 'john@gmail.com');
```

🔥 4. Spring Data JPA

✅ Definition

- Spring Data JPA is NOT a JPA provider.
- It is a Spring framework module that provides an abstraction layer on top of JPA (Hibernate).
- Its main goal: remove boilerplate code for DAOs (Data Access Objects).

✅ Key Features

- Repository abstraction (CrudRepository, JpaRepository).
- Auto-generated queries (e.g., findByName, findByEmail).
- Paging and sorting support.
- Integration with Spring Boot.

✅ Example

Instead of writing:

```
public interface UserRepository extends JpaRepository<User, Integer> {

   List<User> findByName(String name);

}
```

➞ Spring Data JPA automatically generates:

```
SELECT * FROM user WHERE name = ?
```

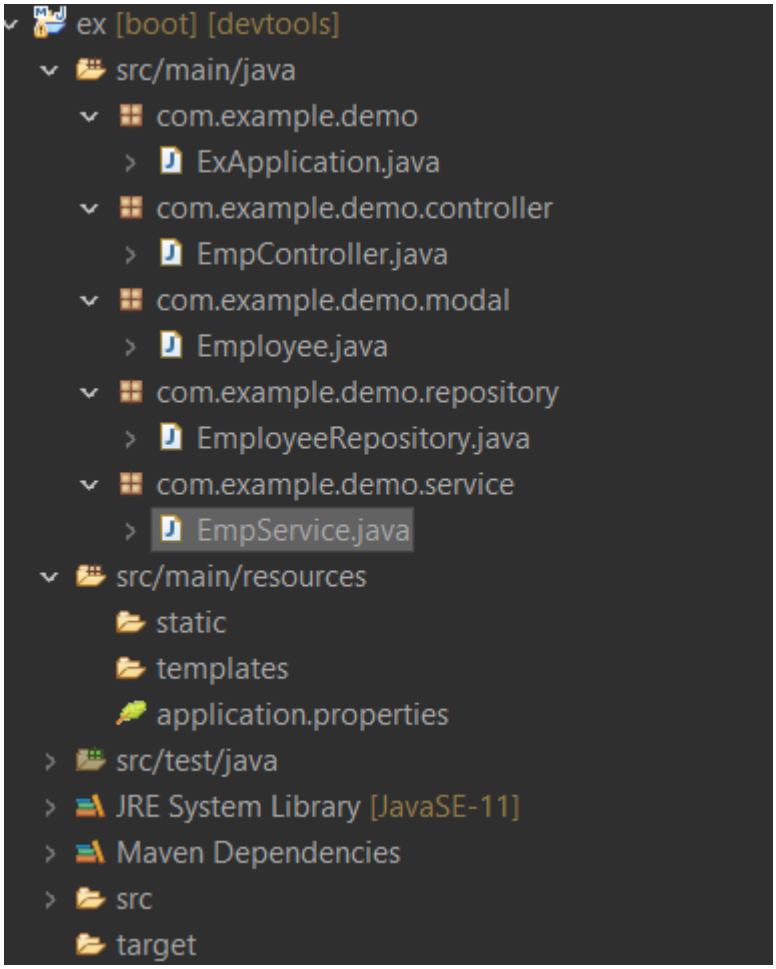
📊 Difference Between MySQL and H2 Database

Database:

Point	MySQL 🐬	H2 Database ⚡
1. Type	Full database server for real applications.	Lightweight database, mainly for testing and development.
2. Installation	Must install and run separately on your system.	Comes with the app, no installation needed.
3. Data Storage	Stores data on disk permanently.	In memory by default (lost when app stops), can also save to file.
4. Speed	Slower for small tasks, better for big data.	Very fast for small datasets or testing.
5. Scalability	Can handle millions of records, suitable for production.	Not for big data or production; best for small tests.
6. Use Case	Production apps, websites, enterprise systems.	Development, unit tests, demos, or prototypes.
7. Spring Boot Setup	Connect via MySQL URL, needs proper DB setup.	Connect via H2 URL, starts/stops with app, has built-in console.

✅ **In practice (with Spring Boot):**

- Use **H2** during development/testing → faster and simpler.
- Use **MySQL** in production → reliable, scalable, and persistent.



Modal layer:

Create a simple POJO(Plain old java class) with some JPA annotation.

- **@Entity: This annotation defines that a class can be mapped to a table**
- **@Id: This annotation specifies the primary key of the entity.**
- **@GeneratedValue: This annotation is used to specify the primary key generation strategy to use. i.e. Instructs database to generate a value for this field automatically. If the strategy is not specified by default AUTO will be used.**

```
package com.example.demo.modal;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

// @Entity annotation defines that a
// class can be mapped to a table
@Entity
public class Employee {

   // @ID This annotation specifies
   // the primary key of the entity.
   @Id

   // @GeneratedValue This annotation
   // is used to specify the primary
   // key generation strategy to use
   @GeneratedValue(strategy = GenerationType.AUTO)
   private long id;
   private String name;
   private String city;
```

```
public Employee() {
   super();
}
public Employee(String name, String city) {
   super();
   this.name = name;
   this.city = city;
}

public String getName() {
   return name;
}

public void setName(String name) {
   this.name = name;
}

public String getCity() {
   return city;
}

public void setCity(String city) {
```

```
        this.city = city;
    }

}
```

DAO(Data access object) layer:

- **@Repository: The @Repository annotation is a marker for any class that fulfills the role or stereotype of a repository (also known as Data Access Object or DAO).**
- **JpaRepository<Employee, Long> JpaRepository is a JPA-specific extension of the Repository. It contains the full API of CrudRepository and PagingAndSortingRepository. So it contains API for basic CRUD operations and also API for pagination and sorting. Here we enable database operations for Employees.**

Why Hibernate Validator?

- **Hibernate Validator** is the **reference implementation of the Bean Validation API (JSR 380)**.
- Spring Boot integrates it seamlessly for validating request bodies, query params, and path variables.

Hibernate Framework

Hibernate ORM (Object Relational Mapping) is a popular Java-based framework used for mapping an object-oriented domain model to a relational database. It provides an efficient way of storing and retrieving data from a database by mapping database tables to Java classes and vice versa.

Features of Hibernate ORM:

1. **Object-Relational Mapping: Hibernate ORM provides support for mapping Java classes to database tables and vice versa. This allows developers to work with objects rather than low-level SQL statements.**
2. **Lazy Loading: Hibernate ORM provides support for lazy loading, which is a technique for loading objects on demand rather than loading them all at once. This can help to improve performance and reduce memory usage.**
3. **Caching: Hibernate ORM provides support for caching, which is a technique for storing frequently accessed data in memory. This can help to improve performance and reduce the number of database queries.**
4. **Transactions: Hibernate ORM provides support for transactions, which are a mechanism for ensuring data consistency and integrity. Transactions can be used to group a set of database operations into a single atomic unit of work.**
5. **Query Language: Hibernate ORM provides support for HQL (Hibernate Query Language), which is a powerful and flexible query language for querying objects. HQL is similar to SQL but operates on objects rather than tables.**

Advantages of using Hibernate ORM Framework:

1. **Simplifies Database Operations: Hibernate ORM simplifies database operations by providing an easy-to-use API for storing and retrieving data from the database. This reduces the amount of boilerplate code required to interact with the database.**
2. **Improves Performance: Hibernate ORM provides support for caching, which improves application performance by reducing the number of database queries. It caches frequently accessed data in memory, which reduces the time taken to access the data from the database.**
3. **Platform Independence: Hibernate ORM is platform-independent, which means it can be used with any relational database. This provides flexibility and makes it easy to switch databases without changing the code.**
4. **Open-Source: Hibernate ORM is an open-source framework, which means it is freely available and can be used by anyone. This reduces the cost of development and makes it easy to get started with Hibernate ORM.**

🔑 CrudRepository vs JpaRepository

1️⃣ CrudRepository

✅ Definition

- CrudRepository is a Spring Data interface that provides basic CRUD (Create, Read, Update, Delete) operations.
- It's the most basic repository abstraction.

✅ Key Methods

```
<S extends T> S save(S entity);        // Create / Update
Optional<T> findById(ID id);           // Read
Iterable<T> findAll();                 // Read all
void deleteById(ID id);                // Delete
void delete(T entity);                 // Delete
long count();                          // Count records
boolean existsById(ID id);             // Check existence
```

👉 Only basic CRUD.

2️⃣ JpaRepository

✅ Definition

- JpaRepository extends CrudRepository and PagingAndSortingRepository.
- It provides JPA-specific methods + support for pagination and sorting.
- Adds extra functionality on top of CrudRepository.

✅ Key Extra Methods

```
List<T> findAll();                     // returns List (not Iterable)
List<T> findAll(Sort sort);            // sorting
Page<T> findAll(Pageable pageable);    // pagination
List<T> findAllById(Iterable<ID> ids); // fetch multiple IDs
```

```
void flush();                                // flush pending changes
<S extends T> List<S> saveAll(Iterable<S>);    // save multiple entities
<S extends T> S saveAndFlush(S entity);        // save + flush
```

👉 Gives you rich querying, sorting, and batch operations.

3️⃣ Hierarchy

```
Repository
   └──── CrudRepository
         └──── PagingAndSortingRepository
               └──── JpaRepository
```

👉 Meaning:

- All JpaRepository methods include CrudRepository + PagingAndSortingRepository.
- So JpaRepository is a superset.

4️⃣ Example

Entity

```
import jakarta.persistence.*;


@Entity
public class User {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;


   private String name;
```

```
    private String email;
}
```

Using CrudRepository

```
import org.springframework.data.repository.CrudRepository;

public interface UserCrudRepo extends CrudRepository<User, Long> {
    // Can still define query methods
    User findByEmail(String email);
}
```

Using JpaRepository

```
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.domain.Pageable;
import org.springframework.data.domain.Page;

public interface UserJpaRepo extends JpaRepository<User, Long> {
    User findByEmail(String email);

    // Extra power: pagination
    Page<User> findByNameContaining(String keyword, Pageable pageable);
}
```

S Example Usage (Controller)

```
@RestController
@RequestMapping("/users")
public class UserController {
```

```
    private final UserJpaRepo repo;

    public UserController(UserJpaRepo repo) {
        this.repo = repo;
    }

    @GetMapping
    public List<User> getAllUsers() {
        return repo.findAll();   // JpaRepository returns List<User>
    }

    @GetMapping("/paged")
    public Page<User> getPagedUsers(@RequestParam int page, @RequestParam int size) {
        return repo.findAll(PageRequest.of(page, size));  // pagination
    }
}
```

6️⃣ When to Use What?

- CrudRepository → When you only need simple CRUD and don't care about pagination/sorting (rare in real apps).
- JpaRepository → Default choice ✅ because it gives CRUD + pagination + batch operations.

7️⃣ Interview-Ready Summary

- CrudRepository → Basic CRUD.

- JpaRepository → Extends CrudRepository, adds pagination, sorting, batch operations, and JPA-specific methods.
- Always prefer JpaRepository in real-world Spring Boot projects because it provides richer functionality with no extra cost.