**✅ Key Points about SpringBoot MVC Application**

**1️⃣ Types of SpringBoot MVC Applications**

* ➔ SpringBoot MVC can be created as:
  1. **Standalone Web Application**
  2. **Deployable Web Application (WAR)**

**2️⃣ Standalone Web Application**

**✅ What Happens:**

A **Standalone Web Application** means the SpringBoot app comes with its own embedded web server (Tomcat) inside the application itself.  
👉 You don’t need to install or configure any external web server.

This type of application is mostly used for development and testing because it is simple and fast to run.

**✅ Why Is It Useful?**

| **Reason** | **Simple Explanation** |
| --- | --- |
| No external setup | You don’t have to manually install or configure Tomcat. |
| Easy to run | Just run the JAR file → Everything works. |
| Good for development | You can quickly make changes and test without extra configuration. |

**✅ How Does It Work Internally?**

When you run this code in the main method:

@SpringBootApplication

public class MyApp {

public static void main(String[] args) {

SpringApplication.run(MyApp.class, args);

}

}

Spring Boot does all these steps automatically:

1. ✅ Starts an embedded Tomcat server.
2. ✅ Creates the IOC Container (to manage beans).
3. ✅ Registers DispatcherServlet (to handle incoming requests).

**✅ Example Scenario**

👉 Example Main Class:

@SpringBootApplication

public class MyApp {

public static void main(String[] args) {

SpringApplication.run(MyApp.class, args);

}

}

When you run:

java -jar myapp.jar

Spring Boot does automatically:

| **Task** | **What Happens** |
| --- | --- |
| Start Embedded Tomcat | Tomcat runs inside the application itself. |
| Create IOC Container | All beans (Controller, Service, DAO) are created and managed by Spring. |
| Register DispatcherServlet | Automatically ready to handle HTTP requests. |

**✅ Why Is This Important?**

Without embedded Tomcat:  
👉 You would have to manually download Tomcat, configure server settings, and deploy WAR files.

With embedded Tomcat:  
👉 You can run the application easily by executing a single command.  
👉 Faster for development and easy to debug.

**✅ Simple Summary of Example**

| **Step** | **Example Action** | **Simple Explanation** |
| --- | --- | --- |
| 1. | Create Main class | Annotate with @SpringBootApplication. |
| 2. | Call run() method | SpringApplication.run(MyApp.class, args); |
| 3. | Everything starts | Embedded Tomcat + IOC + DispatcherServlet all start automatically. |
| 4. | Access URL | Visit http://localhost:8080/index.jsp in browser. |

**3️⃣ Deployable Web Application (WAR)**

**✅ What Happens:**

When you create a SpringBoot MVC application as a **WAR file**, it is meant to run on an **external web server** like Tomcat or JBoss.  
This is commonly used when deploying the app to production, where multiple applications run on the same server.

**✅ Why Do We Need a WAR?**

* ✅ A WAR (Web ARchive) file packages the application so that external web servers (Tomcat, JBoss, etc.) can understand and run it.
* ✅ Unlike a standalone JAR, a WAR does not start an embedded server by itself.

👉 Example Analogy:  
Think of a WAR file like a pre-packed folder ready to be unpacked by the server.

**✅ What Is ServletInitializer and Why Is It Needed?**

* ✅ Normally, SpringBoot creates the IOC Container and DispatcherServlet automatically when using SpringApplication.run().
* ✅ But when deploying WAR in an external server, the server itself starts the application.
* ✅ Therefore, we need a special class called ServletInitializer, which helps the external server know:
  1. How to create the IOC Container
  2. How to register DispatcherServlet

👉 Example ServletInitializer code:

public class ServletInitializer extends SpringBootServletInitializer {

@Override

protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {

return application.sources(MyApp.class);

}

}

**✅ What Does This Code Do?**

| **Line** | **Explanation** |
| --- | --- |
| extends SpringBootServletInitializer | Makes the class able to configure the app when deployed in external server. |
| configure() method | Points to the main SpringBoot configuration class (MyApp.class). |

👉 The external Tomcat server calls this method to know how to start the SpringBoot application.

**✅ Example Scenario – How It Works**

1. Programmer builds the WAR:

mvn clean package

1. The generated WAR file:

myapp.war

1. Programmer copies the WAR file to external Tomcat:

cp myapp.war /apache-tomcat/webapps/

1. External Tomcat automatically calls the ServletInitializer.configure() method to:
   * Create the IOC container
   * Register DispatcherServlet
2. Application becomes accessible via URL:

http://localhost:8080/MyApp/index.jsp

**✅ Why Is This Important?**

Without ServletInitializer, external Tomcat won’t know how to start the SpringBoot app.  
It would throw an error like:

No Spring Boot Application Context found.

This small class is essential for the external server to boot the application properly.

**✅ Simple Summary of Example**

| **Step** | **Example Action** | **Simple Explanation** |
| --- | --- | --- |
| 1. | Build WAR | mvn clean package → creates myapp.war |
| 2. | Deploy WAR | Copy WAR to /webapps/ folder of Tomcat |
| 3. | ServletInitializer helps | External Tomcat calls configure() to create IOC + DispatcherServlet |
| 4. | URL Access | http://localhost:8080/MyApp/index.jsp works perfectly |

## 4️⃣ Shared Concept: IOC Container

### ✅ What Happens:

Whenever you run a SpringBoot MVC application – whether it is a **Standalone Web App (JAR)** or a **Deployable Web App (WAR)** – SpringBoot creates an **IOC Container** called:

AnnotationConfigWebApplicationContext

### ✅ What Is IOC Container?

👉 IOC stands for **Inversion of Control**.

| **Simple Meaning** |
| --- |
| Spring manages objects (beans) for you, instead of you creating objects manually. |

👉 Think of it as a special box where Spring keeps all important objects (like Controllers, Services, DAOs).

### ✅ Example Scenario – Bean Management

Suppose you have a Controller class:

@Controller

public class StudentController {

@Autowired

private StudentService studentService;

}

👉 Normally, you would need to do:

StudentController studentController = new StudentController();

StudentService studentService = new StudentService();

studentController.setStudentService(studentService);

❌ But with IOC Container:  
Spring automatically creates objects:

IOC Container:

Bean -> StudentController

Bean -> StudentService

And injects dependencies automatically → You don’t write new() or set methods.

### ✅ Why Is AnnotationConfigWebApplicationContext Important?

| **Feature** | **Simple Explanation** |
| --- | --- |
| Annotation Based | It reads @Controller, @Service, @Repository, and @Autowired annotations. |
| Automatically Scans | Spring scans the package for annotated classes and creates beans. |
| Manages Bean Lifecycle | It creates, injects, and manages beans without programmer effort. |

👉 Example:  
When the app starts, Spring Boot runs:

AnnotationConfigWebApplicationContext context = new AnnotationConfigWebApplicationContext();

Then it scans:

@Controller → Creates StudentController bean

@Service → Creates StudentService bean

@Repository → Creates StudentDAO bean

### ✅ Simple Summary of Example

| **Step** | **Example Action** | **Simple Explanation** |
| --- | --- | --- |
| 1. | Add annotations | Use @Controller, @Service, @Autowired. |
| 2. | App Starts | SpringBoot automatically creates AnnotationConfigWebApplicationContext. |
| 3. | Beans Created | All required objects are created and managed. |
| 4. | Dependency Injection | Spring injects StudentService into StudentController automatically. |

**5️⃣ Project Structure Example**

**✅ What Happens:**

When you create a SpringBoot MVC project, the folder structure is organized in a specific way to help the application run smoothly.

👉 Example Project:

BootMVCProj1-DisplayingHomePage/

│

├── src/main/java/

│ └── in.ineuron/

│ ├── BootMvcProj1DisplayingHomePageApplication.java (@SpringBootApplication)

│ ├── ServletInitializer.java (extends SpringBootServletInitializer)

│ ├── Controller Classes

│ ├── Service Classes

│ └── DAO Classes

│

├── src/main/resources/

│ ├── application.properties

│ └── templates/ (Thymeleaf or JSP views)

│

├── pom.xml (Maven build configuration)

└── target/ (Compiled files and final JAR/WAR file)

**✅ Why Is This Structure Important?**

| **Folder** | **Purpose** |
| --- | --- |
| src/main/java/ | Holds Java code (Controllers, Services, DAOs, Application config). |
| in.ineuron/ | Package where your Java classes reside. |
| BootMvcProj1DisplayingHomePageApplication.java | Main class with @SpringBootApplication → Starts the SpringBoot app. |
| ServletInitializer.java | Needed if deploying as WAR → Helps external Tomcat start the app. |
| src/main/resources/application.properties | Holds configuration (port, context path, view prefix/suffix, etc.). |
| templates/ or WEB-INF/pages/ | Where JSP/Thymeleaf view files (HTML templates) are stored. |
| pom.xml | Contains project dependencies and build config (for Maven). |
| target/ | Where the JAR/WAR file is created after build. |

**✅ Example Scenario – Simple Explanation**

1. Programmer creates a new SpringBoot MVC project → BootMVCProj1-DisplayingHomePage
2. SpringBoot generates the correct folder structure automatically.

👉 Example of the Main Class:

@SpringBootApplication

public class BootMvcProj1DisplayingHomePageApplication {

public static void main(String[] args) {

SpringApplication.run(BootMvcProj1DisplayingHomePageApplication.class, args);

}

}

👉 Example of Controller Class:

@Controller

public class HomeController {

@RequestMapping("/home")

public String homePage() {

return "home"; // Logical view name

}

}

👉 Example of application.properties:

server.port=8080

spring.mvc.view.prefix=/WEB-INF/pages/

spring.mvc.view.suffix=.jsp

👉 Example of View (home.jsp):

<html>

<body>

<h1>Welcome to Home Page</h1>

</body>

</html>

**✅ Why Is This Structure Useful?**

* ✅ Organizes code in a standard way → Easy to understand and maintain.
* ✅ Separates concerns:  
  → Controllers handle requests  
  → Services handle business logic  
  → DAOs handle database  
  → Views handle UI rendering
* ✅ application.properties centralizes configuration → Easy to change settings (like port, context path) without modifying code.

**✅ Simple Summary of Example**

| **Step** | **Example Action** | **Simple Explanation** |
| --- | --- | --- |
| 1. | Create Project | Standard folder structure is generated automatically. |
| 2. | Write Controller | Handle HTTP request → Return Logical View Name. |
| 3. | Configure application.properties | Set view prefix, suffix, server port. |
| 4. | Place Views | JSP files in /WEB-INF/pages/. |
| 5. | Build App | Run mvn clean package → creates JAR/WAR file. |

**6️⃣ SpringApplication.run() Auto Tasks (for Standalone App)**

**✅ What Happens:**

When you run a **Standalone SpringBoot MVC Application**, you call this method:

SpringApplication.run(MyApp.class, args);

👉 This single line of code makes SpringBoot automatically do many important tasks in the background, without you needing to do anything extra.

**✅ What Are These Automatic Tasks?**

| **Task** | **What Happens (Simple Explanation)** |
| --- | --- |
| ✅ Embedded Tomcat | SpringBoot starts the Tomcat server inside the app. You don’t need to install or configure Tomcat manually. |
| ✅ IOC Container | SpringBoot creates an object manager called AnnotationConfigWebApplicationContext that manages your beans (Controller, Service, DAO). |
| ✅ DispatcherServlet | Automatically created and registered in the embedded Tomcat. It handles incoming web requests. |
| ✅ Error Filters | Special components created automatically to handle common errors (like 404 Page Not Found). No manual error handling setup needed. |
| ✅ Other Beans | SpringBoot scans for classes annotated with @Controller, @Service, @Repository, etc., and creates these beans automatically, with dependencies injected. |

**✅ Example Scenario – How It Works**

👉 Programmer writes just this main class:

@SpringBootApplication

public class MyApp {

public static void main(String[] args) {

SpringApplication.run(MyApp.class, args);

}

}

Without doing anything extra, SpringBoot does:

1. ✅ Starts Embedded Tomcat → App is ready to accept HTTP requests.
2. ✅ Creates IOC Container → Manages all beans automatically.
3. ✅ Registers DispatcherServlet → Automatically handles HTTP routing.
4. ✅ Creates Error Filters → Handles errors like 404 without programmer effort.
5. ✅ Creates and Injects Beans →  
   Example:
   * StudentController created as a bean
   * StudentService created as a bean
   * StudentDAO created as a bean  
     Dependencies like @Autowired StudentService are injected automatically.

**✅ Why Is This Important?**

| **Reason** | **Simple Explanation** |
| --- | --- |
| Saves Time | No need to manually configure Tomcat or DispatcherServlet. |
| Reduces Errors | No risk of forgetting to configure important components. |
| Focus on Business Logic | Programmer only writes Controller, Service, DAO → Spring handles the rest. |

**✅ Simple Summary of Example**

| **Step** | **Example Action** | **Simple Explanation** |
| --- | --- | --- |
| 1. | Write Main Class | Add @SpringBootApplication and call SpringApplication.run(). |
| 2. | Embedded Tomcat Starts | Automatically handles HTTP requests. |
| 3. | IOC Container Created | Manages beans automatically. |
| 4. | DispatcherServlet Registered | Automatically routes requests to the correct controller method. |
| 5. | Error Filters Auto-Created | Handles errors like 404 without extra code. |
| 6. | Beans Auto-Created | Controllers, Services, DAOs are automatically available. |

**7️⃣ Context Path Configuration**

**✅ What Happens:**

By default, when a SpringBoot MVC app runs, there is **no context path configured**.  
👉 This means the application runs at the root of the server, so URLs look like:

http://localhost:8080/index.jsp

But sometimes, we want to run the application under a specific **context path** to group URLs, avoid conflicts, or organize multiple apps.

**✅ How Do We Configure Context Path?**

👉 In application.properties, add this entry:

server.servlet.context-path=/FirstApp

spring.mvc.view.prefix=/WEB-INF/pages/

spring.mvc.view.suffix=.jsp

server.port=9999

| **Property** | **Meaning** |
| --- | --- |
| server.servlet.context-path=/FirstApp | Sets the context path to /FirstApp. |
| spring.mvc.view.prefix=/WEB-INF/pages/ | Where view files (JSP) are stored. |
| spring.mvc.view.suffix=.jsp | All views have .jsp suffix. |
| server.port=9999 | App listens on port 9999 instead of default 8080. |

**✅ Example Scenario – How It Works**

Before configuration:

* Accessing the home page:

http://localhost:8080/index.jsp

After configuration:

* Accessing the home page becomes:

http://localhost:9999/FirstApp/index.jsp

👉 Example Explanation:

* You configured context path as /FirstApp.
* Port is changed to 9999 → so now the base URL becomes:

http://localhost:9999/FirstApp/

👉 What Happens Internally:

1. SpringBoot reads application.properties automatically during startup.
2. It sets the context path to /FirstApp.
3. Embedded Tomcat now knows the app is accessible under /FirstApp.

**✅ Why Is This Important?**

| **Benefit** | **Simple Explanation** |
| --- | --- |
| Avoid Conflicts | Multiple apps can run on same server with different context paths. |
| Organized URLs | Makes URLs meaningful → Example: /FirstApp/home, /SecondApp/about. |
| Easier Deployment | Helps in production where multiple apps are deployed on the same server. |

**✅ Simple Summary of Example**

| **Step** | **Example Action** | **Simple Explanation** |
| --- | --- | --- |
| 1. | Add entry in application.properties | server.servlet.context-path=/FirstApp |
| 2. | Start the App | Embedded Tomcat reads config and sets context path. |
| 3. | Access URL | Use: |

http://localhost:9999/FirstApp/index.jsp

``` |

| 4. | App Runs Under Context | All URLs are now prefixed with `/FirstApp`. |

👉 This way, configuring context path becomes simple and fully automatic, without extra coding.