

➤ How Spring Boot Works?

- ⌘ **dependencies** are nothing but **.jar** files which contains the *classes* that we want to use.
- ⌘ **maven** just download the **JARs**, and puts them on the **compile + runtime classpath**.
- ⌘ **modules** are **JARs** with extra responsibilities.
 - ⌘ `spring-web`, `spring-context`, `spring-jdbc`, `spring-data-jpa`
- ⌘ When you run your application, **JVM** searches for all the **.class** files inside the **classpath** which is provided by **maven** (I.e. **classpath** list is provided by Maven).
 - ⌘ **target** folder is one of the classpaths; but it is not the only classpath;
 - ⌘ **.jar** files present in the External Libraries are stored in **local Maven Cache**. Which is not inside the project.
 - * `~/m2/repository/org/springframework/spring-webmvc/6.1.x/spring-webmvc-6.1.x.jar`
- ⌘ Then how **JVM** gets to know about the **.class** files that will be used?
 - ⌘ **Maven** creates a **classpath** where all the list of paths are present.
 - ⌘ **JVM** sees this and use those classes.
- ⌘ **External Libraries** are nothing but the **classpaths** that are not inside your project.
- ⌘ When you create **.jar** file of your application, the **self created classes** and the **external libraries classes** will be stored separately.

```
myapp.jar
├─ BOOT-INF/classes   ← your classes
├─ BOOT-INF/lib       ← dependency JARs
```

- ⌘ **classes** will contain the **self created classes**.
 - ⌘ **lib** will contain the **external libraries classes**.
- **Maven vs JVM**
- ⌘ **maven** works at **build time** and **JVM** works at **run time**.
 - ⌘ Initially **maven** is run when you trigger the commands like `mvn test`, `mvn compile`, `mvn package`, `mvn spring-boot:run` .
 - ⌘ It downloads the dependencies after reading the **pom.xml** and stores them in `~/m2/repository` (*maven cache path*).
 - ⌘ Then it **compiles** the **.java** files and convert those to **.class** files (bytecodes) (*if the command that was executed was `mvn compile`*)

- ⌘ Depending upon the maven command executed, it'll do the thing.
 - ⌘ One important thing: **maven** is also written in **java**. So it also needs one **JVM** to run it. It is called **Maven JVM** (its not any special JVM, just normal JVM only)
 - ⌘ Inside this Maven JVM, **maven modules** are being used, not **spring modules**.
 - ⌘ After finishing its work, Maven creates files on disk (JAR, class files). Later, a separate JVM loads those files and runs the Spring Boot application..
- ⌘ Now **JVM** part comes.
 - ⌘ **JVM** never reads **pom.xml**, it is just read by **maven** to download the dependencies and provide those to **JVM**.
 - ⌘ **JVM** **use** those dependencies to run the application.
- There is one plugins that is present in the **pom.xml** which is “**maven-compiler-plugin**”.
 - ⌘ **maven-compiler-plugin** invokes **javac** behind the scenes, passing all the required flags, paths, and options.
 - ⌘ Needful flags means
 - ⌘ -classpath → where dependencies are
 - ⌘ -processorpath → where annotation processors (like Lombok) are
 - ⌘ -source / -target (or --release) → Java version
 - ⌘ -d target/classes → where compiled .class files go
 - ⌘ list of .java source files
- **spring-boot-maven-plugin**
 - ⌘ **spring-boot-maven-plugin** does NOT run your application logic.
 - ⌘ It prepares and packages your Spring Boot application so it can be run easily by the JVM.
 - ⌘ Without this the dependencies won't be there in the **.jar** file, you would have to copy and paste those dependencies to run the application.
 - ⌘ Also it copies the **tomcat jars** into that.

➤ **Spring Boot Work-Flow**

- ⌘ Dependencies are JAR files that contain reusable classes.
- ⌘ Maven reads pom.xml, downloads dependencies into ~/.m2/repository, and manages build steps.
- ⌘ Plugins are used by Maven to compile, test, package, and run the application.
- ⌘ Maven helps prepare classpath information, but the JVM actually uses the classpath to load .class files.
- ⌘ IntelliJ shows dependency JARs from Maven cache under External Libraries.
- **Runtime Flow**
 - ⌘ JVM starts and executes main()
 - ⌘ SpringApplication.run() processes @SpringBootApplication
 - ⌘ Auto-configuration classes listed in the AutoConfiguration.imports file are considered
 - ⌘ Beans are created only if conditions match, using conditional annotations
- **Maven vs JVM**
 - ⌘ Maven works at build time
 - ⌘ JVM works at run time
 - ⌘ Maven downloads and prepares dependency JARs
 - ⌘ JVM loads classes from those JARs and runs the application
- **Plugins**
 - ⌘ maven-compiler-plugin
 - ⌘ Invokes javac with required flags
 - ⌘ Handles annotation processing (Lombok)
 - ⌘ spring-boot-maven-plugin
 - ⌘ Packages the app into an executable fat JAR
 - ⌘ Bundles dependencies (including embedded Tomcat)
 - ⌘ Makes java -jar app.jar possible

- In the **plugins** sections, under **maven-compiler-plugin**, there is something called **annotationProcessorPaths**

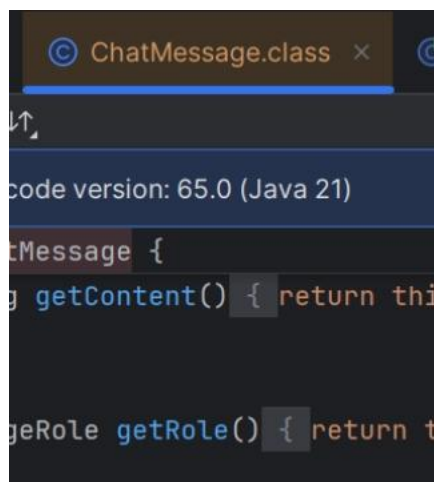
```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <configuration>
    <annotationProcessorPaths>
      <path>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
      </path>
    </annotationProcessorPaths>
  </configuration>
</plugin>
```

- ♣ **annotationProcessorPaths** tells javac which **JARs** contain **annotation processors** and should be loaded during **compilation**.
- ♣ Lombok must be available on the annotation processor path for its annotations (@Getter, @Setter, etc.) to work.
- ♣ Annotation processors are **compile-time** tools that analyze annotations and modify or generate code before **.class** files are created.
- ♣ The order of entries in annotationProcessorPaths does not control execution order; processors do not run sequentially.
- In case of **lombok** and **MapStruct**, why **lombok-mapstruct-binding** is needed?
 - ♣ Lombok **does not generate new classes**; it **modifies existing classes** at **compile time** by altering the compiler's internal representation (AST: Abstract Syntax Tree).
 - ♣ MapStruct generates new mapper classes and relies on the annotation-processing API to inspect methods.
 - ♣ Lombok's AST modifications are not fully visible to MapStruct by default.
 - ♣ **lombok-mapstruct-binding** acts as a bridge, making **Lombok-generated getters and setters visible to MapStruct** during annotation processing.
 - ♣ **This problem is about visibility, not execution order.**
 - ♣

- Also you can see below, in this plugin **lombok** is included in the **excludes** list

```
<plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
  <configuration>
    <excludes>
      <exclude>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
      </exclude>
    </excludes>
  </configuration>
</plugin>
```

- ⌘ This is because, **lombok** doesn't provide any **.class** files that JVM needs to run the application.
- ⌘ It just injects the **getters** and **setters** to the existing classes during **compile-time**.
- ⌘ When you see the **.class** files, you'll see the **getters** and **setters** method's implementations there; **lombok** runs during **compilation-time**; as the **getters** and **setters** are already generated, hence there is no need of including **lombok** in the **.jar** file.



How Auto-Configuration works?

➤ The imports file inside the auto-configure jar is read

- ♣ Spring reads this file:

```
♣ META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports
```

- ♣ This file contains a list of auto-configuration classes, like

```
♣ org.springframework.boot.autoconfigure.web.servlet.DispatcherServletAutoConfiguration  
♣ org.springframework.boot.autoconfigure.orm.jpa.HibernateJpaAutoConfiguration
```

➤ Spring loads those classes, and evaluates their annotations.

➤ Then it checks the conditional annotations

- ♣ Each auto-config class has things like the below:

```
♣ @ConditionalOnClass(DataSource.class)  
♣ @ConditionalOnMissingBean(DataSource.class)  
♣ @ConditionalOnProperty(...)
```

- ♣ Spring checks:

- ♣ * Is the required class present? Is the required property set? Is a user-defined bean already present?

- ♣ If all the conditions pass, then only bean is auto-configured.

➤ If the bean already exists, it will NOT create one

- ♣ Lets say we have manually configured one bean, like

```
♣ @Bean  
♣ public DataSource myDataSource() { ... }
```

- ♣ Then the below condition will fail

```
♣ @ConditionalOnMissingBean(DataSource.class)
```

- ♣ Because this bean is already created.

➤ Overall flow

```
♣ @SpringBootApplication  
  ↓  
  @EnableAutoConfiguration  
  ↓  
  Read AutoConfiguration.imports  
  ↓  
  Load AutoConfig classes  
  ↓  
  Check @Conditional annotations  
  ↓  
  If conditions match → create beans  
  Else → skip
```

- So, in simple terms: **Spring Boot auto-configuration works by scanning predefined auto-configuration classes and conditionally creating beans based on classpath presence, configuration properties, and existing user-defined beans.**
- Write **debug=true** inside the *application.properties* file to see the beans being auto-configured.
-

➤ Spring Web

➤ Servlet:

- ♣ It's a Java class that receives HTTP requests and sends HTTP response.
- ♣ Browser → Servlet → Business Logic → Servlet → Browser

➤ Tomcat

- ♣ Tomcat is a **web server + servlet container**.
- ♣ A Servlet is a Java class that Tomcat loads, manages, and calls when an HTTP request arrives.
- ♣ Tomcat does 2 jobs:
 - ♣ Listens for HTTP requests (like a normal web server)
 - ♣ Runs **Servlets** to handle those requests.
- ♣ So it has 2 parts:
 - ♣ **HTTP Server + Servlet Container**

➤ Servlet

- ♣ **Servlet** is just a normal Java class that **extends HttpServlet**.
- ♣ HttpServlet has methods like doGet(), doPost() ..etc

```
public class MyServlet extends HttpServlet {  
    protected void doGet(HttpServletRequest req, HttpServletResponse res) {  
        res.getWriter().write("Hello");  
    }  
}
```

➤ A servlet does nothing on its own; unless and until *tomcat* calls it.

➤ How **Tomcat** calls the servlets?

- ♣ It maintains a mapping table from **URL pattern** to **Servlet instance**

URL Pattern	Servlet Instance
/hello	MyServlet
/login	LoginServlet

- ♣ There are basically **3** ways to create this mapping:

- ♣ **@WebServlet**

```
@WebServlet("/hello")  
public class MyServlet extends HttpServlet {}
```


- **web.xml** (old way)

```
<servlet>
    <servlet-name>myServlet</servlet-name>
    <servlet-class>com.MyServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>myServlet</servlet-name>
    <url-pattern>/hello</url-pattern>
</servlet-mapping>
```

- Programmatic (Spring Boot style)

```
* ServletRegistrationBean
```

- Lets say you trigger a GET request on the URL: <http://localhost:8080/hello> . Then the following steps happens:

- Browser sends HTTP request
- Tomcat receives request
- Tomcat checks URL mapping table
- Finds: "/hello" → MyServlet
- Tomcat creates (or reuses) MyServlet instance
- Calls doGet() method
- Response returned to browser

- **Tomcat calls your servlet — not Spring**

- **NOTE: The servlet instance is created by Tomcat; not by YOU; not by Spring;**

- You can just create the class extending HttpServlet; that's it;

```
@Component
@WebServlet("/hello")
public class MyServlet extends HttpServlet {}
```

- Even if you write like this it'll not work; because here the **instance** (i.e. **bean**) is created by Spring; not Tomcat;
- **✗ Spring DOES NOT scan @WebServlet by default**
- **✗ Tomcat DOES NOT scan Spring beans**

- You can say why to write **@Component**, just write **@WebServlet("/hello")** and move on.

- Because we want **tomcat** to notice this not **spring**.

⌘ But, in Spring Boot application, tomcat is handled by Spring.

➤ 2 scenarios are there:

⌘ Traditional Tomcat (war deployment)

```
Tomcat (standalone)
├ Scans classpath
├ Finds @WebServlet
└ Creates servlet
```

⌘

⌘ @WebServlet works automatically

⌘ No Spring involvement needed

⌘ Spring Boot (Embedded Tomcat)

```
Spring Boot Application
├ Starts embedded Tomcat
└ Tomcat waits for Spring instructions
```

⌘

⌘ ! Tomcat does NOT scan for @WebServlet on its own

⌘ ! It only registers what Spring explicitly tells it to

➤ What happens in Tomcat in Spring Boot Application?

- Key points which is TRUE everytime

- ♣ Tomcat always creates servlet instances.

- ♣ Spring NEVER creates servlet instances for Tomcat to use.

- In traditional Tomcat, it *itself* scans for @WebServlet and create instance of those and map.

- But in Spring Boot application, Spring embeds Tomcat inside itself;

- ♣ Spring itself start that Tomcat server.

- ♣ And in here, Tomcat doesn't scan for the @WebServlet, rather it waits for Spring to give it instruction to do the scan (if we write @ServletComponentScan), after that Tomcat scans and registers those servlets.

```
Spring Boot starts
↓
Spring creates embedded Tomcat
↓
Tomcat waits for instructions
↓
Spring tells Tomcat what to register
```

```
@ServletComponentScan
@SpringBootApplication
public class SpringSecurityApplication {
```

```
@WebServlet("/hello") no usages
public class HelloServlet extends HttpServlet {
```

- ♣ No need to write @Component

-

➤ Creating Custom Servlet extending HttpServlet class

➤ HttpServlet

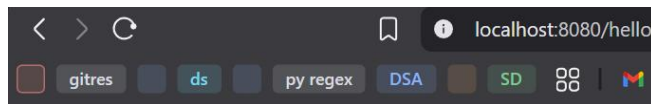
⌘ You can see, all the methods inside **HttpServlet** only add some **error content** in the response.

⌘ **NOTE:** the method doesn't return anything; they do have **request** and **response** objects, they just add some content to the **response** object.

```
protected void doPost(HttpServletRequest req, HttpServletResponse resp) thro
    String msg = lStrings.getString( key: "http.method_post_not_supported");
    this.sendMethodNotAllowed(req, resp, msg);
}
```

⌘ If you see here, it is just adding some **error** to the response via the method **sendMethodNotAllowed**.

⌘ HttpServlet is the base class which is to be extended to write custom servlet. But you need to override the method and implement by your own; otherwise error response will be visible.



Whitelabel Error Page

This application has no explicit mapping for /error, so you are s

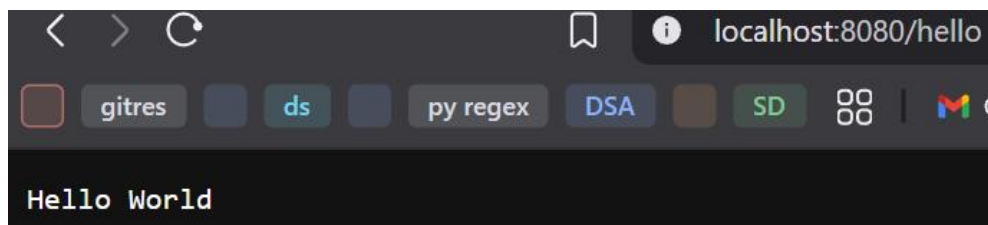
Thu Dec 25 21:26:49 IST 2025

There was an unexpected error (type=Method Not Allowed, sta
HTTP method GET is not supported by this URL

⌘ I overridden the **service** method and added in the response

```
public void service(HttpServletRequest req, HttpServletResponse res)
    res.getWriter().println("Hello World");
}
```

⌘ Now the response is coming in the browser.



- ⌘ You can set the content type of the response as well.

```
public void service(HttpServletRequest req, HttpServletResponse res)
    res.setContentType("text/html");
    res.getWriter().println("<h1><b>Hello World</b></h1>");
}
```

- You can also manually configure the Tomcat without the help of Spring Boot

```
Tomcat tomcat = new Tomcat();
tomcat.setPort(8080);

Context context = tomcat.addContext(contextPath: "", docBase: null);
Tomcat.addServlet(context, servletName: "HelloServlet", new HelloServlet());
context.addServletMappingDecoded(pattern: "/hello", name: "HelloServlet");

tomcat.start();
tomcat.getServer().await();
```

- ⌘ **SpringApplication.run** is not there. And also **@WebServlet** is not written in HelloServlet class.



➤ How all the mappings are being done in Spring Boot Application?

- As we know, **Tomcat** has a mapping table where the **end points** are mapped to **servlets**, according to the end-point, **tomcat** calls the **specific servlets**.

- But in case of Spring Boot Application, only one servlet is there which is **DispatcherServlet** (it also extend *HttpServlet* only); it is auto-configured.

♣ **Tomcat** maps **DispatcherServlet** for all the end-points which is “/*”

♣ After that, **DispatcherServlet** calls the controllers to update the *response object*.

♣ **Tomcat** doesn't know about the controllers at all.

➤

➤ F

➤ Ff

➤ F

➤ F

➤ F

➤ F

➤ F

➤ f