1.6 Auto configuration, Application Context and Internal working of a Spring Boot Application

pom.xml

- Maven is a popular build *automation* tool used in many java projects. In a spring boot project ***dependencies*** are specified in ***pom.xml*** file. Meven then resolves theses **dependencies** and includes them in **class path**.
- *Staters* like *spring-boot-stater-parent* include a **ton** of **third-party libraries** into your **project** – by default. Its *AutoConfiguration* use these dependencies to **set-up** and **pre-configure** these libraries automatically.

What is Auto Configuration?

- *Autoconfiguration* refers to the **mechanism** that **automatically** configures *spring applications* bases on the **dependencies** present on the **class path** and **other application specific settings**.
- This feature simplifies the *setup* and *development* process, allowing developers to focus more on writing ***business logic*** rather than ***configuring the framework***.

How Autoconfiguration works?

Classpath Scanning	**Spring boot** scans the **class path** for the presence of certain libraries and classes. Based on what it finds, it applies corresponding **configurations**.
Configuration Classes	**Spring boot** contains a numerous **autoconfiguration** classes, each responsible for **configuring a specific part** of application.
Conditional Beans	Each ***autoconfiguration class*** uses ***conditional checks*** to decide if it should be applied. These conditions include the ***presence of specific classes***, the ***absence of user define beans***, and ***specific property settings***.

Core features of AutoConfiguration:

- *@PropertySources Auto-Registration*
  When you run the main method of Spring boot application, Spring boot will automatically register *17* of *PropertySources* for you.
- *Enhanced conditional support*
  Spring Boot provides extra ***@Conditional*** annotations to control ***bean creation*** based on specific conditions, making configuration more **flexible**:
  - ***@ConditionalOnBean(DataSource.class):*** Loads the bean only if a ***DataSource bean*** already exists.
  - ***@ConditionalOnClass(DataSource.class):*** Loads the bean only if ***DataSource*** is present on the ***classpath.***
  - ***@ConditionalOnProperty("my.property"):*** Loads the bean only if the property ***my.property*** is defined in *application.properties*.

What happens when we start a Spring Boot application:

1. *JVM* starts and main() method is executed
2. *SpringApplication.run(MyApp.class, args)*
3. Spring Boot detects:
   - *@SpringBootApplication* on *MyApp.class*
     - Which includes:
       ├── *@SpringBootConfiguration* → marks as *@Configuration*
       ├── *@ComponentScan* → scans current package & subpackages
       └── *@EnableAutoConfiguration* → loads *auto-configs* from *spring.factories*
4. Creates a *SpringApplication* object:
   - Sets up *ApplicationContext*
   - Sets up *Environment*
   - Registers Listeners, Initializers
5. *ApplicationContext* is created
6. *Environment* is prepared (e.g., *application.properties*, *args*, *profiles*)
7. *Beans* are scanned and registered via *@ComponentScan* and *Auto-configured beans* are *registered* (e.g., DataSource, WebServer)
8. *ApplicationContext* is *refreshed*:
   - All *beans* are created
   - *Dependencies* are *@Autowired*
   - *Lifecycle* methods are called
9. *Embedded Web Server* (e.g., *Tomcat*) is started
10. *CommandLineRunner* or *ApplicationRunner beans* (if any) are executed
11. *Application* is now fully started and ready to *serve requests*