



Spring Boot framework



Spring Boot is a framework built on top of the Spring Framework that simplifies the development of Java-based applications. It aims to make the process of setting up and running Spring applications faster and easier by providing a range of built-in features and configurations.

Spring Boot is designed to handle and simplify the usage of core Spring components like Spring Core, Spring Data, and others

First Step

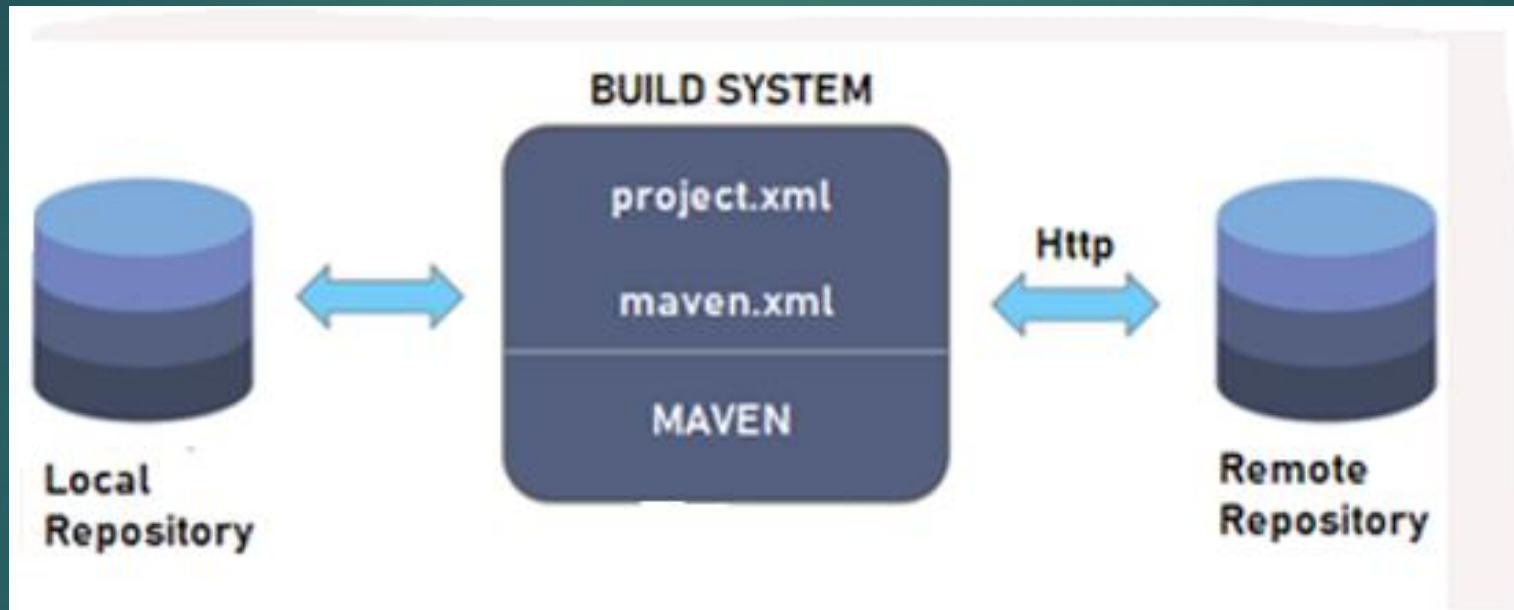
you need to add lib file to use tool you hope to use

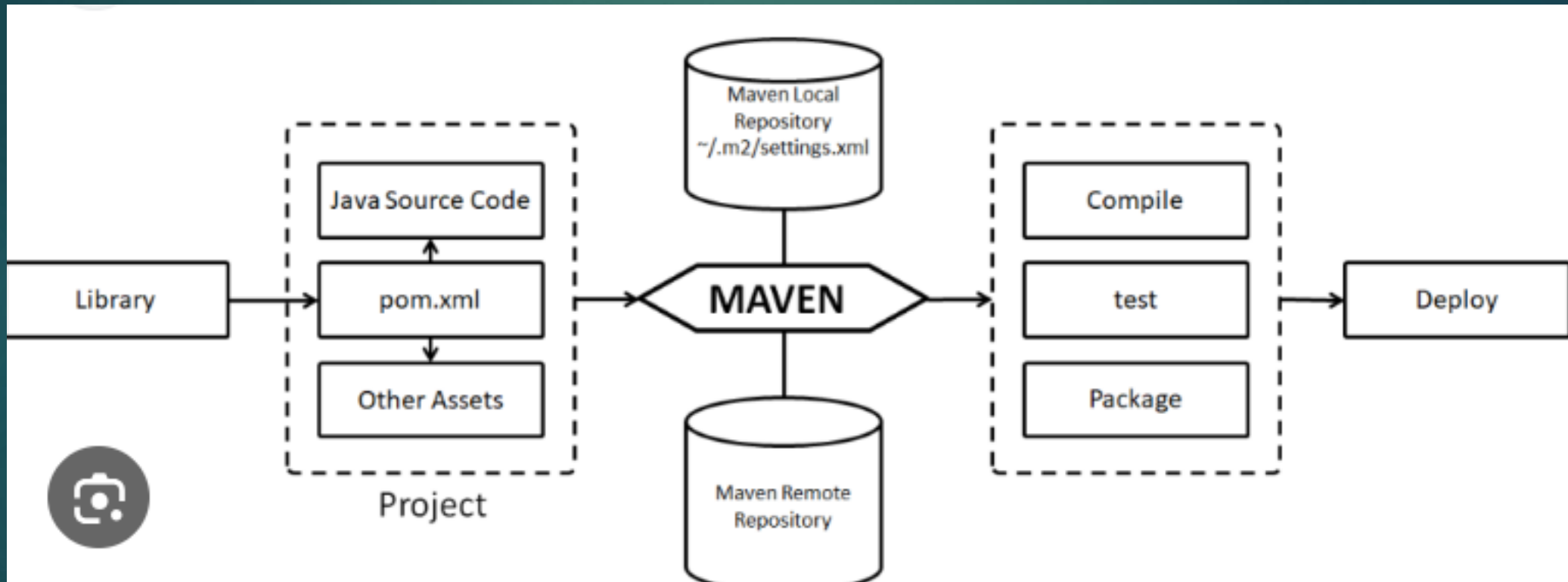
A white square containing the text "no!" in red and "no!" in black, stacked vertically.



**we will use maven
so you need to
know what is
maven**

Maven is a super tool used in Java-based projects. It simplifies project setup, dependency management, and builds by providing a standardized way to manage a project's lifecycle.





```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>2.5.4</version>
  </dependency>
</dependencies>
```


1. `<dependencies>` :

- This tag is used to define all the **dependencies** that your project requires. Dependencies are external libraries that your project needs to function correctly. In this case, we have only one dependency: `spring-boot-starter-web`.


2. `<dependency>` :

- This tag defines a single dependency. Each dependency requires at least three key elements:
 - `groupId`
 - `artifactId`
 - `version`

3. `<groupId>` :

- The `groupId` uniquely identifies the project or organization that provides the dependency. In this case:

xml

 Copy code


```
<groupId>org.springframework.boot</groupId>
```

- `org.springframework.boot` : This indicates that the dependency is part of the Spring Boot framework, which is a project under the organization "Spring" (more specifically, the Spring Boot module).

4. `<artifactId>`:

- The `artifactId` is the unique name for the dependency within the specified group. In this case:

xml

 Copy code


```
<artifactId>spring-boot-starter-web</artifactId>
```

- `spring-boot-starter-web`: This refers to a Spring Boot "starter" dependency, which is a pre-configured set of libraries for building web applications using Spring MVC. It includes everything needed to create a web application, such as:
 - Spring MVC (Model-View-Controller framework)
 - An embedded web server (like Tomcat or Jetty)
 - JSON serialization via Jackson

5. `<version>`:

- The `version` tag specifies the exact version of the dependency that you want to use. In this case:

xml

 Copy code

```
<version>2.5.4</version>
```

- `2.5.4`: This specifies that version 2.5.4 of `spring-boot-starter-web` will be used. Maven will ensure that this exact version is downloaded and included in your project.

1- **spring-boot-starter-web**

This dependency includes everything needed to build web applications, such as Spring MVC, embedded Tomcat, and Jackson for JSON processing.

2- **spring-boot-starter-data-jpa**

If you need to interact with a database, the spring-boot-starter-data-jpa is commonly used to simplify database interactions using JPA (Java Persistence API) and Hibernate as the ORM.

3. Oracle Driver SQL
driver for dp

Let's Create Project



`@SpringBootApplication` is an annotation in the Spring Boot framework that is used to mark the main class of a Spring Boot application. It enables several essential Spring features and simplifies the configuration by combining three annotations into one:

1. `@SpringBootConfiguration`: Equivalent to `@Configuration`, it tells Spring that this class contains configuration settings and beans for the application context.
2. `@EnableAutoConfiguration`: This instructs Spring Boot to automatically configure your application based on the dependencies present in your classpath. It saves developers from having to manually configure things like database connections, web servers, or messaging systems.
3. `@ComponentScan`: This tells Spring to scan the current package and its sub-packages for Spring components like controllers, services, and repositories, and automatically register them in the application context.

API (Application Programming Interface) is a set of rules and tools that allows one software application to interact with another. It defines the methods and data formats that applications can use to communicate with each other, enabling them to request and exchange information or services.



Spring Boot framework



Endpoints: Specific URLs or addresses where the API is accessible. Each endpoint typically corresponds to a specific functionality (e.g., fetching user data, sending a message).

Requests and Responses: The API client (e.g., your app) sends a request to the server, ..and the server responds with the data or confirmation of the requested action

API Method

GET: Retrieve data.

POST: Send data to create a new resource.

PUT: Update existing data.

DELETE: Remove data.

```
# spring
server.port=8081

# Oracle Database Configuration
spring.datasource.driver-class-name=oracle.jdbc.OracleDriver
spring.datasource.url=jdbc:oracle:thin:@//localhost:1521/orclpdb
spring.datasource.username=hr
spring.datasource.password=hr

# JPA and Hibernate Configuration for Oracle
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true|
```

@Repository

`@Repository` is a Spring annotation used to mark a class as a **Data Access Object (DAO)** or **Repository**, which is responsible for handling the interaction with the database. It essentially acts as a bridge between your application and the persistence layer (e.g., a relational database like MySQL, PostgreSQL).

@Service

The `@Service` annotation in Spring is used to mark a class as a **Service** component in the application's service layer. It indicates that the class contains business logic and acts as an intermediary between the controller layer (which handles HTTP requests) and the repository/data access layer (which interacts with the database).

@Controller

`@Controller` is a Spring annotation used to define a **controller** in a Spring MVC (Model-View-Controller) application. A controller is responsible for handling HTTP requests, processing the request data, and returning the appropriate view (usually HTML, JSON, etc.) to the client.

In Spring Boot (or any Java-based application with an MVC architecture), creating a **DTO (Data Transfer Object)** for your `Student` entity can be useful for several reasons:

1. **Separation of Concerns:** The entity (`Student`) is often a direct representation of the database table. Exposing it directly to the API or UI might lead to tight coupling between the database schema and the external layers (controllers, views). DTOs help separate what you expose from your internal data model.
2. **Security:** You may not want to expose all fields of your `Student` entity to the outside world. For example, sensitive data such as passwords or internal system fields (e.g., creation date, internal IDs) should not be part of the response. DTOs allow you to control which fields are exposed.
3. **Efficiency:** Entities may contain large amounts of data, some of which may not be necessary for certain requests or responses. With DTOs, you can optimize the data being transferred, especially when interacting with external services or clients.
4. **Data Transformation:** Sometimes, the data coming from the entity needs to be transformed or formatted differently for the client. For example, you might want to aggregate, calculate, or format the data differently in the DTO before sending it out.
5. **Validation:** DTOs can also help with input validation. When receiving data from a client, you might validate fields differently from how you would treat them in your database. A DTO allows for separation between database logic and client-side input validation.



Spring Boot framework

