Commands:

mvn spring-boot:run

Build Lifecycle Basics

Maven is based around the central concept of a build lifecycle. What this means is that the process for building and distributing a particular artifact (project) is clearly defined.

For the person building a project, this means that it is only necessary to learn a small set of commands to build any Maven project, and the [POM](https://maven.apache.org/guides/introduction/introduction-to-the-pom.html) will ensure they get the results they desired.

There are three built-in build lifecycles: default, clean and site. The default lifecycle handles your project deployment, the clean lifecycle handles project cleaning, while the site lifecycle handles the creation of your project's web site.

A Build Lifecycle is Made Up of Phases

Each of these build lifecycles is defined by a different list of build phases, wherein a build phase represents a stage in the lifecycle.

For example, the default lifecycle comprises of the following phases (for a complete list of the lifecycle phases, refer to the [Lifecycle Reference](https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html#Lifecycle_Reference)):

* validate - validate the project is correct and all necessary information is available
* compile - compile the source code of the project
* test - test the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed
* package - take the compiled code and package it in its distributable format, such as a JAR.
* verify - run any checks on results of integration tests to ensure quality criteria are met
* install - install the package into the local repository, for use as a dependency in other projects locally
* deploy - done in the build environment, copies the final package to the remote repository for sharing with other developers and projects.

These lifecycle phases (plus the other lifecycle phases not shown here) are executed sequentially to complete the default lifecycle. Given the lifecycle phases above, this means that when the default lifecycle is used, Maven will first validate the project, then will try to compile the sources, run those against the tests, package the binaries (e.g. jar), run integration tests against that package, verify the integration tests, install the verified package to the local repository, then deploy the installed package to a remote repository.

**🌱 IoC Container in Spring Boot (Inversion of Control)**

**✅ What is IoC Container?**

In **Spring Boot**, the **IoC Container** (also called **Spring Container**) is the **core** part that:

* Creates objects (beans)
* Manages their lifecycle
* Injects dependencies automatically
* Serves the beans wherever needed

It **inverts control** from your code creating objects ➝ to the container managing it.

**✅ Types of IoC Containers**

Spring provides two main containers:

* **BeanFactory**: Basic container with lazy loading.
* **ApplicationContext**: More powerful (used in Spring Boot).

In Spring Boot, **ApplicationContext is used by default**.

**✅ How @Component Makes a Bean**

@Component

public class MyService {

public void serve() {

System.out.println("Service Called!");

}

}

* The @Component annotation marks this class as a **Spring-managed component (bean)**.
* During application startup, Spring scans the classpath for such annotated classes and **registers them as beans** in the IoC container.

**✅ How Are Beans Scanned?**

Spring Boot uses **Component Scanning** via @ComponentScan.

By default, Spring Boot starts scanning from the package of your main class (the class with @SpringBootApplication):

@SpringBootApplication // includes @ComponentScan

public class MyApp {

public static void main(String[] args) {

SpringApplication.run(MyApp.class, args);

}

}

@ComponentScan looks for:

* @Component
* @Service
* @Repository
* @Controller

All are **specializations of @Component**.

📌 You can customize scanning like:

@ComponentScan(basePackages = "com.example.services")

**✅ How Are Beans Put in the Container?**

1. On startup, Spring Boot performs classpath scanning.
2. All classes annotated with @Component (or its variants) are:
   * **Instantiated**
   * **Managed** as singletons by default
3. Beans are **registered** inside the **ApplicationContext (container)** with a bean ID.

**✅ How Are Beans Served During Runtime?**

There are 2 main ways:

**1. Autowired by Type**

@Component

public class MyController {

@Autowired

private MyService myService;

public void handle() {

myService.serve();

}

}

Spring injects the correct MyService bean automatically.

**2. Constructor Injection (Recommended)**

@Component

public class MyController {

private final MyService myService;

@Autowired // optional since Spring 4.3+

public MyController(MyService myService) {

this.myService = myService;

}

public void handle() {

myService.serve();

}

}

**✅ Lifecycle Summary**

| **Step** | **Action** |
| --- | --- |
| 1️⃣ | Application starts |
| 2️⃣ | Spring scans classes with @Component |
| 3️⃣ | Instances are created and dependencies resolved |
| 4️⃣ | Beans are registered in ApplicationContext |
| 5️⃣ | Beans are injected wherever needed via @Autowired |
| 6️⃣ | You access beans via controller, service, etc. |

**✅ Final Note**

🔁 You don’t need to manually create or manage objects in Spring Boot — the IoC container handles it.

This makes your code:

* Cleaner
* Decoupled
* Testable
* Easier to maintain

**@SpringBootApplication**

This annotation is only added once in the main class (class containing main method)



The @SpringBootApplication annotation is the **main entry point** of a Spring Boot application. It combines several important annotations into **one shortcut**, enabling component scanning, auto-configuration, and bean definition.

**✅ 1. What is @SpringBootApplication?**

It is a **meta-annotation** defined as:

@Target(TYPE)

@Retention(RUNTIME)

@Documented

@Inherited

@SpringBootConfiguration

@EnableAutoConfiguration

@ComponentScan

public @interface SpringBootApplication {

...

}

This means it **includes three core annotations** that drive Spring Boot behavior.

**✅ 2. Breakdown of Annotations**

**🔹 @SpringBootConfiguration**

* It’s a specialized version of @Configuration
* Tells Spring that this class contains **bean definitions**
* Used to **bootstrap the application context**

@Configuration

public class AppConfig {

@Bean

public MyService myService() {

return new MyService();

}

}

So @SpringBootConfiguration is simply:

@Target(TYPE)

@Retention(RUNTIME)

@Documented

@Configuration

public @interface SpringBootConfiguration {

}

**🔹 @EnableAutoConfiguration**

* Enables Spring Boot’s **auto-configuration feature**
* Automatically configures beans based on the **classpath**, **properties**, and **defined beans**

Example:

* If spring-boot-starter-web is in classpath, it auto-configures:
  + DispatcherServlet
  + Tomcat
  + Jackson, etc.

You can exclude unwanted auto-configs:

@SpringBootApplication(exclude = { DataSourceAutoConfiguration.class })

**🔹 @ComponentScan**

* Tells Spring to **scan the package** of your main class and its sub-packages
* It finds @Component, @Service, @Repository, @Controller etc., and registers them as beans

You can also manually specify packages:

@ComponentScan(basePackages = "com.example.services")

**✅ 3. How It Works Internally**

When you run:

@SpringBootApplication

public class MyApp {

public static void main(String[] args) {

SpringApplication.run(MyApp.class, args);

}

}

Here’s what happens:

1. SpringApplication.run():
   * Starts the **Spring IoC container (ApplicationContext)**
   * Performs component scanning via @ComponentScan
   * Loads beans defined by @SpringBootConfiguration
   * Applies auto-configurations via @EnableAutoConfiguration
2. **Beans get created**, autowired, and ready to serve your app.

**✅ 4. Summary Table**

| **Annotation** | **Purpose** |
| --- | --- |
| @SpringBootConfiguration | Marks the class as a config source |
| @EnableAutoConfiguration | Automatically configures based on dependencies |
| @ComponentScan | Scans packages to register beans |

**✅ 5. Visual Flow**

@SpringBootApplication

↓

-------------------------------

| @SpringBootConfiguration | → ApplicationContext setup

| @EnableAutoConfiguration | → Auto-configures based on classpath

| @ComponentScan | → Scans and registers your beans

-------------------------------

↓

SpringApplication.run(...) → App boots up

**✅ @Configuration and @Bean – The Core of Manual Bean Creation in Spring**

**🔹 What is @Configuration?**

* It marks a class as a **source of bean definitions**.
* It's a specialization of @Component, so the class itself is picked up in **component scanning**.
* It's used to **define beans manually** via methods annotated with @Bean.

@Configuration

public class AppConfig {

@Bean

public MyService myService() {

return new MyService(); // Spring will call this method and register the returned object as a bean

}

}

**🔹 What is @Bean?**

* Used on methods inside a @Configuration class to **define a bean manually**.
* Spring calls these methods at startup and **registers the return object as a bean in the IoC container**.

@Bean

public MyRepository myRepository() {

return new MyRepositoryImpl();

}

**🔧 How It Works Internally**

When Spring processes a @Configuration class:

1. It creates a **proxy** of the configuration class (CGLIB proxy).
2. It calls each @Bean method **only once** and stores the result as a singleton (by default).
3. If any other bean calls the same method, Spring returns the **same object** (not a new one) from the container.

This avoids duplicate objects:

@Bean

public A a() {

return new A();

}

@Bean

public B b() {

return new B(a()); // reuses the same A bean, doesn't call a() again

}

**🧠 Why Use @Configuration + @Bean Instead of @Component?**

| **Use-case** | **Recommendation** |
| --- | --- |
| For **manual control** over bean creation or customization | ✅ Use @Configuration + @Bean |
| For **auto-detected**, simple components | ✅ Use @Component, @Service, etc. |

**📌 Summary**

| **Annotation** | **Role** |
| --- | --- |
| @Configuration | Declares the class contains bean definitions |
| @Bean | Declares a method that returns a bean object |

So in simple terms:

@Configuration lets Spring know to look inside the class, and @Bean tells it **what to create and manage** in the container.

**REST**

**🌐 What is REST / RESTful API?**

**✅ 1. REST – Representational State Transfer**

**REST** is an architectural style for designing **networked applications**, especially **web services**.

A **RESTful API**:

* Uses **HTTP methods** (GET, POST, PUT, DELETE, etc.)
* Treats **URLs as resources**
* Is **stateless**: Each request is independent and contains all necessary info
* Returns **data in JSON or XML**

**✅ 2. HTTP Methods Mapping**

| **HTTP Method** | **Action** | **Description** |
| --- | --- | --- |
| GET | Read | Fetch a resource or list |
| POST | Create | Add a new resource |
| PUT | Update | Replace a resource |
| PATCH | Partial Update | Modify part of a resource |
| DELETE | Delete | Remove a resource |

**✅ 3. Example REST API for "Student"**

| **Action** | **URL** | **Method** |
| --- | --- | --- |
| Get all students | /students | GET |
| Get one student | /students/1 | GET |
| Create student | /students | POST |
| Update student | /students/1 | PUT |
| Delete student | /students/1 | DELETE |

**🚀 REST in Spring Boot**

Spring Boot makes building REST APIs **easy and fast** using:

* @RestController
* @RequestMapping
* @GetMapping, @PostMapping, etc.
* @RequestBody, @PathVariable, etc.

**✅ Step-by-Step: Creating a Simple REST API**

**1️⃣ Create a model class**

public class Student {

private int id;

private String name;

// constructor, getters, setters

}

**2️⃣ Create a REST Controller**

import org.springframework.web.bind.annotation.\*;

import java.util.\*;

@RestController

@RequestMapping("/students")

public class StudentController {

private List<Student> studentList = new ArrayList<>();

@GetMapping

public List<Student> getAllStudents() {

return studentList;

}

@PostMapping

public String addStudent(@RequestBody Student student) {

studentList.add(student);

return "Student added!";

}

@GetMapping("/{id}")

public Student getStudentById(@PathVariable int id) {

return studentList.stream()

.filter(s -> s.getId() == id)

.findFirst()

.orElse(null);

}

@DeleteMapping("/{id}")

public String deleteStudent(@PathVariable int id) {

studentList.removeIf(s -> s.getId() == id);

return "Student deleted!";

}

}

**✅ Important Annotations**

| **Annotation** | **Purpose** |
| --- | --- |
| @RestController | Tells Spring this class handles REST requests. Combines @Controller + @ResponseBody |
| @RequestMapping | Base URL path for the controller |
| @GetMapping | Maps GET HTTP requests |
| @PostMapping | Maps POST requests |
| @PathVariable | Binds URL path segment to method param |
| @RequestBody | Binds request JSON body to Java object |

**✅ JSON Example**

POST /students

Content-Type: application/json

{

"id": 1,

"name": "Arbaz"

}

**✅ Summary**

| **Concept** | **Spring Boot Feature** |
| --- | --- |
| REST Principles | Stateless, resource-based, HTTP methods |
| Expose REST API | @RestController, @RequestMapping, @\*Mapping |
| Data handling | JSON via @RequestBody and @ResponseBody |
| Auto config | Spring Boot auto-configures Jackson, DispatcherServlet, etc. |

**RestController, RequestMapping, PathVariable, RequestBody etc.**

**🌐 @RestController in Spring Boot**

**✅ What is @RestController?**

* It is a **specialized controller** used to **create RESTful web services**.
* Combines:
* @Controller + @ResponseBody
* It returns **data (usually JSON)** directly in the HTTP response body instead of rendering a view.

**✅ Example:**

@RestController

public class HelloController {

@GetMapping("/hello")

public String sayHello() {

return "Hello, World!";

}

}

📌 This will return plain "Hello, World!" as HTTP response, not a view.

**❓ Should all endpoints be public?**

Yes ✅. In Spring Boot, all handler methods (mapped with @GetMapping, @PostMapping, etc.) **must be public** because:

* They are **called reflectively** by the Spring framework.
* private or protected methods **won’t be accessible** for request mapping.

**🔁 @RequestMapping**

**✅ What is it?**

* The **base annotation** used to map web requests to methods or classes.
* Can be used to **handle all HTTP methods** (GET, POST, PUT, etc.)

@RequestMapping("/api")

@RestController

public class MyController {

@RequestMapping("/hello")

public String greet() {

return "Hi!";

}

}

🔸 Modern alternative:  
Use shortcut annotations like:

* @GetMapping
* @PostMapping
* @PutMapping
* @DeleteMapping

**✅ Example with all:**

@RestController

@RequestMapping("/users") // base path

public class UserController {

@GetMapping("/{id}")

public String getUser(@PathVariable int id) {

return "User ID: " + id;

}

@PostMapping

public String addUser(@RequestBody String user) {

return "User added: " + user;

}

}

**🔀 @PathVariable**

**✅ Purpose:**

* Binds a **URL path value** to a method parameter.

**✅ Example:**

@GetMapping("/students/{rollNo}")

public String getStudentByRoll(@PathVariable int rollNo) {

return "Student Roll: " + rollNo;

}

📌 Request: GET /students/101  
➡️ Output: Student Roll: 101

**📦 @RequestBody**

**✅ Purpose:**

* Converts the **HTTP request body (usually JSON)** into a Java object.
* Requires a converter like **Jackson** (auto-configured in Spring Boot).

**✅ Example:**

@PostMapping("/students")

public String createStudent(@RequestBody Student student) {

return "Added: " + student.getName();

}

📌 Request:

POST /students

Content-Type: application/json

{

"id": 1,

"name": "Arbaz"

}

➡️ Output: Added: Arbaz

**🔚 Summary Table**

| **Annotation** | **Purpose** |
| --- | --- |
| @RestController | Marks class to handle REST requests; returns JSON |
| @RequestMapping | Maps URL to controller/method (generic) |
| @GetMapping / @PostMapping | HTTP method-specific mapping |
| @PathVariable | Extracts value from URL path |
| @RequestBody | Converts JSON request body into Java object |

**🔍 What is MongoDB?**

**MongoDB** is a **NoSQL, document-oriented database**. It stores data in a **flexible, JSON-like format called BSON** (Binary JSON), making it schema-less, fast, and scalable. Unlike traditional relational databases (RDBMS) like **MySQL**, MongoDB does not use tables and rows. Instead, it uses **collections** and **documents**.

**🆚 MongoDB vs MySQL (Relational DB)**

| **Feature** | **MongoDB (NoSQL)** | **MySQL (Relational)** |
| --- | --- | --- |
| **Data Storage Format** | BSON (Binary JSON) | Tables with rows and columns |
| **Schema** | Schema-less | Schema-based (strict structure) |
| **Relationships** | Embedded or referenced documents | Joins with foreign keys |
| **Scalability** | Horizontal (sharding supported) | Vertical (scale-up typically) |
| **Transactions** | Limited (multi-doc supported since 4.0) | Fully supported |
| **Best Use Cases** | Unstructured, hierarchical data | Structured, relational data |
| **Query Language** | MongoDB Query Language (MQL) | Structured Query Language (SQL) |

**🧱 MongoDB Key Terminologies (With MySQL Analogy)**

| **MongoDB Term** | **Description** | **Relational DB Equivalent** |
| --- | --- | --- |
| **Database** | Holds a set of collections | Database |
| **Collection** | A group of MongoDB documents | Table |
| **Document** | A record in MongoDB, stored as BSON | Row |
| **Field** | A key-value pair in a document | Column |
| **Embedded Doc** | Document nested inside another document | Related table (joined) |
| **ObjectId** | Unique ID automatically generated | Primary Key (auto-incr) |
| **Index** | Improves search/query performance | Index |

**📦 Example Comparison:**

**MySQL Table (Users)**:

| id | name | age |

|----|-------|-----|

| 1 | Arbaz | 25 |

**MongoDB Document in "users" Collection**:

{

"\_id": ObjectId("..."),

"name": "Arbaz",

"age": 25

}

**🧾 Basic MongoDB Commands You Must Know**

**📁 Database Related**

show dbs # List all databases

use <dbname> # Switch/create database

db # Show current database

**📂 Collection Related**

show collections # List collections in current DB

db.createCollection("users") # Create a collection

db.users.drop() # Delete a collection

**📄 Document Related**

db.users.insertOne({ name: "Arbaz", age: 25 }) # Insert one document

db.users.insertMany([{...}, {...}]) # Insert multiple documents

db.users.find() # Show all documents

db.users.findOne() # Find a single document

db.users.find({ age: { $gt: 20 } }) # Query with condition

**✏️ Update & Delete**

db.users.updateOne({ name: "Arbaz" }, { $set: { age: 26 } })

db.users.deleteOne({ name: "Arbaz" })

**📌 Other Useful Commands**

db.users.countDocuments() # Count documents

db.users.find().sort({ age: 1 }) # Sort ascending by age

db.users.find().limit(5) # Limit results

db.users.updateMany(...) # Update multiple docs

db.users.deleteMany(...) # Delete multiple docs

**🧠 When to Use MongoDB?**

* When you have **unstructured, nested, or frequently changing data**.
* Ideal for **real-time analytics**, **content management**, **IoT**, **social media**, and **mobile apps**.
* When horizontal scalability and high availability are important.

**Spring AI**

The video titled "Why Spring AI" discusses the importance of integrating AI features into applications using Spring AI as a bridge. It highlights how Spring AI simplifies managing connections with various large language model (LLM) providers, such as OpenAI, Anthropic, and Azure OpenAI.

The speaker outlines that while you can connect directly to these providers, using Spring AI offers several advantages, particularly in terms of abstraction. This is crucial because different LLM providers come with unique SDKs and coding requirements, which can complicate application development when working with multiple models.

For example, OpenAI provides a straightforward Java SDK, while Anthropic's SDK is still in beta and differs in structure. The lecture emphasizes the need for abstraction to simplify future updates and maintenance. By adopting Spring AI, developers can keep their code clean and manageable, ultimately enhancing modern application development when dealing with various AI services.

**MCP**

Model Context Protocol (MCP)

Model Context Protocol (MCP) is an open protocol that enables LLMs to access external tools and data sources. For more details about MCP, see the MCP documentation.

Vector embeddings

Cosine similarity between two words: Closer to 1 means similar, closer to zero means opposite

Can be helpful for semantic search. If we searching and exact thing/word is not present then it will look similar/close to the searched word.