Getting Started - Master Spring Framework and Spring Boot

19 June 2025 23:07

Introduction to Spring and Spring Boot

Top Java Frameworks

- **Spring** and **Spring Boot** are the two leading frameworks in the Java ecosystem.

Challenges for Beginners

- **First steps** with Spring and Spring Boot can be difficult.
  - **Why?** There is a lot of new terminology:
    - Dependency Injection
    - Inversion of Control (IoC)
    - Autowiring
    - Auto Configuration
    - Starter Projects
    - (and many more)
- These frameworks can be used to build:
  - Web applications
  - REST APIs
  - Full-stack applications (with React)
  - Integrate with many tools/frameworks: Maven, Gradle, Spring Data, JPA, Hibernate, Docker, Cloud, etc.

Course Approach

- The course provides a **simple path** to learn Spring, Spring Boot, and related tools.
  - Focus on **fundamentals** of each framework/tool.
  - **Hands-on** learning approach.
  - **More than 10 projects** using Maven and Gradle.
  - Designed for **absolute beginners**.
  - **Goal:** Make it easy to start with Spring and Spring Boot.

Learning Recommendations

1. **Active Learning**
   - Think and take notes as you go through the course.
   - Write down interesting or important points.
2. **Regular Review**
   - After a few sections, step back and review presentations and videos.
   - Reflect on what you've learned.

- Active engagement and review increase long-term retention.

Course Structure – Three-Pronged Approach

1. **Videos & Presentations**
   - Introduce key concepts.
2. **Hands-on Projects**
   - Build multiple real projects to reinforce learning.
3. **Quizzes**
   - Designed to reinforce and test your understanding.

- This approach is intended to help you remember content for a long time.

Additional Recommendations

- **It's not a race:** Take your time; focus on deep understanding.
- **Replay videos** as needed.
- **Most important: Have fun!**
  - The course is designed to be interactive and hands-on to keep learning enjoyable.

Instructor's Closing

- Excited to help you learn Spring and Spring Boot.
- Looking forward to the next step in your learning journey.

Summary Table

Topic	Details
Main Frameworks	Spring, Spring Boot
Initial Challenge	Steep learning curve, lots of terminology
Application Types	Web Apps, REST APIs, Full Stack (with React), Integration with tools
Tools/Frameworks	Maven, Gradle, Spring Data, JPA, Hibernate, Docker, Cloud, etc.
Learning Path	Simple, fundamentals-focused, hands-on, beginner-friendly
Course Features	10+ projects, quizzes, video presentations
Learning Tips	Take notes, review regularly, be actively involved
Recommendations	Don't rush, replay videos, have fun

Step-by-Step Guide to Installing Java and Eclipse IDE:

1. **Access the GitHub Repository:**
   - Go to the homepage of the course's GitHub repository.
   - https://github.com/in28minutes/master-spring-and-spring-boot/#installing-tools
   - Scroll down to the "Installing Tools" section.
2. **Install Java:**
   - Find the video guides for installing Java, available for different operating systems (Windows, Mac, Linux).
   - Watch the video that matches your operating system and follow the steps to install Java.

- Make sure to install at least Java 17, especially if you plan to use Spring Boot 3 or newer.
- There's also a troubleshooting video if you face any installation issues.

3. **Install Eclipse IDE for Enterprise Java Developers:**
   - Below the Java section, you'll find links and videos to install Eclipse IDE, tailored for various operating systems.
   - Choose the correct video for your OS and follow the instructions to complete the installation.
   - If you run into any problems, refer to the troubleshooting guide provided.
4. **Use the Latest Versions:**
   - It's recommended to always use the latest versions of both Java and Eclipse for best compatibility with course materials and Spring Boot.
5. **Note on Java Installation:**
   - The installation process for Java has remained consistent across recent versions (Java 16, 17, 18, etc.) and is expected to stay the same.

Once both tools are installed, you're ready for the next step in the course! If you encounter any issues, check the troubleshooting resources in the GitHub repository.

Getting Started with Java Spring Framework - 1 - Create Maven Projects

18 June 2025 16:30

Application Architectures & Key Frameworks

1. **Evolution of Application Architectures:**
   - Over the past 20 years, application architectures have evolved from basic web applications to web services, REST APIs, full stack applications, microservices, and now, cloud deployments.
2. **Frameworks Used:**
   - Many frameworks are used to build modern applications, including:
     - Spring, Spring Boot, Spring MVC, Hibernate, Spring Security, Spring Data, Spring Cloud, and others.
3. **Two Most Important Frameworks to Learn:**
   - **1. Spring Framework:**
     - Essential for building maintainable applications.
     - Provides key features like Dependency Injection and auto-wiring.
   - **2. Spring Boot:**
     - Makes using the Spring Framework much easier.
     - Reduces setup and configuration code, increasing productivity.
4. **Productivity Example:**
   - Before Spring, setting up a production-ready application could take ~1,000 lines of code.
   - With Spring Framework, this reduces to about 700 lines.
   - With Spring Boot, it's even less—around 400 lines.
5. **Course Focus:**
   - This module will focus mainly on the Spring Framework.
   - Many beginners find Spring's terminology confusing (e.g., tight coupling, loose coupling, dependency injection, IoC container, ApplicationContext, Spring beans, autc-wiring, ComponentScan).
6. **Learning Approach:**
   - The instructor, Ranga Karanam, will use his experience to make Spring easy to understand.
   - The course will explain concepts with simple, hands-on examples.

In short:
The two most important frameworks for building great Java applications are Spring Framework and Spring Boot. Spring helps you build maintainable code with powerful features, and Spring Boot makes the process even easier and faster. The course will guide you through these concepts step by step.

Section Goals and Approach for Learning Spring Framework

1. **Types of Applications You Can Build:**
   - With Java, Spring, and Spring Boot, you can build:
     - Web applications
     - REST APIs

        - Full stack applications
        - Microservices
        - Many other types of applications
2. **Importance of Spring Framework:**
    - Spring provides all the core features needed for any type of Java application.
    - Understanding Spring helps you learn Spring Boot easily.
    - Knowing Spring makes it easier to debug problems.
3. **Main Goal of This Section:**
    - To understand the core features of the Spring Framework using a hands-on, practical approach.
4. **Project You'll Build:**
    - You'll create a simple "Hello World Gaming" app using modern Spring techniques.
    - The app will let you run classic games like Mario, SuperContra, and Pacman.
5. **Learning Approach – Iterative Steps:**
    - The module will use an iterative approach to build and improve the application:
        1. **Iteration 1:**
            - Build tightly coupled Java code (everything directly connected).
            - Create a GameRunner class to run different game classes.
        2. **Iteration 2:**
            - Introduce loose coupling using Java interfaces.
            - Create a GamingConsole interface, and have each game implement this interface.
            - The instructor will explain interfaces if you're not familiar.
        3. **Iteration 3:**
            - Bring in the Spring Framework for loose coupling (level 1).
            - Use Spring to create and manage objects (Spring Beans) and handle wiring.
        4. **Iteration 4:**
            - Go further with loose coupling using Spring annotations (level 2).
            - Spring will fully manage creating, wiring, and auto-wiring objects.
6. **Key Terminology You'll Learn:**
    - Tight coupling, loose coupling, IoC (Inversion of Control) container, application context, component scan, dependency injection, Spring Beans, auto-wiring.
7. **Support for Beginners:**
    - Some terms might seem difficult at first, but the instructor will explain everything step by step, making it easy to understand by the end of the module.

In short:
You'll learn the core ideas of Spring Framework by building a simple gaming application, improving it step by step. You'll start with basic Java code and gradually introduce interfaces and Spring features, so you'll understand both the "why" and "how" of key Spring concepts in a practical, hands-on way.

Step-by-Step Guide: Creating and Importing a Spring Project

1. **Go to Spring Initializr:**
    - Visit start.spring.io (the Spring Initializr website).
2. **Project Setup:**
    - **Project:** Select **Maven Project** as the build tool.
    - **Language:** Choose **Java**.
    - **Spring Boot Version:**
        - Use the latest **Released Version** of Spring Boot 3 (e.g., 3.0.1, 3.1.0, etc.).
        - Avoid Snapshot or RC (Release Candidate) versions unless there's no released version available.

- **Group ID :** Enter com.in28minutes
- **Artifact ID:** Enter learn-spring-framework
- Similar to Package Name and Class Name.
- **Java Version:** Choose Java 17 or the latest available.

3. **Generate the Project:**
   - Click **Generate**.
   - A ZIP file with your project will be downloaded.
4. **Unzip the Project:**
   - Extract the downloaded ZIP file to a folder on your computer (e.g., C:/learn-spring-framework).
5. **Open Eclipse IDE:**
   - Launch Eclipse IDE for Enterprise Java and Web Developers.
   - Choose a workspace directory.
6. **Import the Project into Eclipse:**
   - Go to **File** > **Import**.
   - Type **Maven** in the search box.
   - Select **Existing Maven Projects** and click **Next**.
   - Browse and select the root directory of your unzipped project.
   - You should see the pom.xml file, with com.in28minutes as the Group ID and learn-spring-framework as the Artifact ID.
   - Click **Finish** to complete the import.
7. **Wait for Import to Complete:**
   - The first import might take 5-10 minutes, especially if you're using a new version of Spring Boot.
   - If you encounter issues, ensure you're using the latest version of Eclipse IDE.
8. **Project Structure:**
   - After import, you'll see the following folders in your project:
     - src/main/java (for source code)
     - src/main/resources (for configuration files)
     - src/test/java (for test code)

In short:
You used Spring Initializr to create a new Spring Boot project with Maven and Java, unzipped it, and imported it into Eclipse IDE. Now you're ready to start coding in your new Spring project!

Summary: Spring Initializr and Maven

1. **What is Spring Initializr?**
   - Spring Initializr is a website (start.spring.io) that helps you create Spring projects quickly and easily.
   - You simply visit the site, make a few choices (like project type, language, and dependencies), and download a ready-to-use project as a zip file.
   - Before Spring Initializr, setting up a new Spring or Spring Boot project could take 1–2 hours. Now, it takes just a few minutes.
2. **How to Use Spring Initializr:**
   - Go to the website, make your selections, download the zip file, extract it, and import it into your IDE.
   - This process is much faster and easier than manual setup.
3. **Maven as Dependency Management Tool:**
   - When creating the project, you choose Maven as your dependency management tool.
   - Maven is a popular tool in the Java ecosystem that manages project dependencies (like Spring Framework).
   - Maven automatically downloads all necessary libraries (such as Spring) and makes them available to your project.
   - You will learn more about Maven in detail later in the course.

4. **Why Use Maven?**
    - Maven handles the "magic" of fetching and managing the libraries your project needs, so you don't have to do it manually.
5. **Best Practices for Version Selection:**
    - Always use the latest **released version** of Spring Boot.
    - Released versions are stable and reliable (e.g., 3.2.8, 3.3.2).
    - Avoid using **snapshot** or **milestone** versions, as they are under development and may not be stable.

In short:
Spring Initializr makes creating Spring projects fast and easy, and Maven automatically manages all your dependencies. Always use the latest released version for stability. You'll use these tools throughout the course to create and manage your Spring projects.

Getting Started with Java Spring Framework - 2 - Setup Java Gaming Application

19 June 2025 23:15

Step-by-Step Summary: Building Your First Game Runner Application in Java

1. **Introduction and Goal:**
   - You're going to write Java code to simulate running classic games like Mario, Super Contra, and Pac-Man.
   - To start, you'll focus on running the Mario game using a special class called GameRunner.
2. **Create the Main Application Class:**
   - In Eclipse, right-click the package com.in28minutes.learnspringframework.
   - Choose **New > Class**.
   - Name the class AppGamingBasic.Java.
   - Add a public static void main(String[] args) method to serve as the entry point.
3. **Write the Main Logic:**
   - Inside the main method:
     - Create a new instance of MarioGame.
     - Create a new instance of GameRunner, passing the MarioGame instance to its constructor.
     - Call the run() method on the GameRunner object to start the game.

```
public class AppGamingBasic {

	public static void main(String[] args) {
		var marioGame = new MarioGame();
		var gameRunner = new GameRunner(marioGame);
		gameRunner.run();

	}

}
```

4. **Fix Compilation Errors by Creating Needed Classes:**
   - Since MarioGame and GameRunner don't exist yet, Eclipse will show errors.
   - Use Eclipse's quick fix (Ctrl+1 or Cmd+1) to **create the MarioGame class** in the package com.in28minutes.learnspringframework.game.
   - Repeat to **create the GameRunner class** in the same package.
5. **Add a Constructor to GameRunner:**
   - In GameRunner, add a field to hold a reference to the game (MarioGame game).

- Create a constructor that takes a MarioGame object and assigns it to the field.

6. **Add the run() Method:**
   - In GameRunner, create a public void run() method.
   - In this method, use System.out.println to print out a message indicating the game is running (e.g., "Running game: [game]").
7. **Run the Application:**
   - Back in AppGamingBasicJava, ensure there are no more compilation errors.
   - Right-click the class and select **Run As > Java Application**.
   - Check the console output. You should see a message like:
     1Running game com.in28minutes.learnspringframework.game.MarioGame@...
   - Congratulations! You have successfully created and run your first version of the game runner.
8. **What's Next:**
   - The instructor suggests that in the next step, you'll add methods to the MarioGame class to simulate actions like pressing left, right, up, and down buttons.

In short:
You created a basic Java application that can run the Mario game using a GameRunner class. You learned how to create classes, constructors, and methods, and saw your code in action in the console. Next, you'll add more features to make the game runner more interactive!

Complete Code:

AppGamingBasic.java
```
package com.in28minutes.learn_spring_framework;

import com.in28minutes.learn_spring_framework.game.GameRunner;
import com.in28minutes.learn_spring_framework.game.MarioGame;

public class AppGamingBasic {

	public static void main(String[] args) {
		var marioGame = new MarioGame();
		var gameRunner = new GameRunner(marioGame);
		gameRunner.run();

	}

}
```

MarioGame.java
```
package com.in28minutes.learn_spring_framework.game;

public class MarioGame {

}
```

GameRunner.java
```
/**
```

```
 *
 */
package com.in28minutes.learn_spring_framework.game;

/**
 *
 */
public class GameRunner {
	MarioGame game;

	public GameRunner(MarioGame game) {

		this.game = game;
	}

	public void run() {
		System.out.println("Running game: " + game);

	}
}
```

Output:

```
Running game: com.in28minutes.learn_spring_framework.game.MarioGame@279f2327
```

Step-by-Step Summary: Adding Button Actions to the Mario Game

1. **Enhance the MarioGame Class:**
   - Add methods in the MarioGame class to represent game actions:
     - public void up() (e.g., Mario jumps)
     - public void down() (e.g., Mario goes into a hole)
     - public void left() (e.g., Mario goes back)
     - public void right() (e.g., Mario accelerates)
   - Each method uses System.out.println to describe the action.
2. **Update the GameRunner Class:**
   - In the run() method of GameRunner, after printing that the game is running:
     - Call each of the action methods on the game object in order:
       - game.up()
       - game.down()
       - game.left()
       - game.right()
   - This simulates pressing each button on the game controller.
3. **Run the Application:**
   - Right-click the main application class (AppGamingBasicJava) and run it as a Java application.
   - Open the console to view the output.
4. **Expected Console Output:**
   - You should see:

     ```
     Running game: com.in28minutes.learn_spring_framework.game.MarioGame@
     279f2327
     Jump
     ```

```
Go into a hole
Go back
Accelerate
```

5. **Summary of Progress:**
   - You have written simple Java code to:
     - Create a game runner and a Mario game.
     - Define specific actions for the Mario game.
     - Use the game runner to trigger those actions.
6. **Key Concept Introduced:**
   - This design is an example of *tightly coupled* code, meaning the GameRunner depends directly on the MarioGame class.
   - The next step will explore why tight coupling can be a problem and how to improve it.

In short:
You added up, down, left, and right actions to your Mario game and used the GameRunner to call these actions. This setup works, but is tightly coupled—something you'll learn more about (and how to improve) in the next steps!

AppGamingBasic.java

```java
package com.in28minutes.learn_spring_framework;

import com.in28minutes.learn_spring_framework.game.GameRunner;
import com.in28minutes.learn_spring_framework.game.MarioGame;

public class AppGamingBasic {

	public static void main(String[] args) {
		var marioGame = new MarioGame();
		var gameRunner = new GameRunner(marioGame);
		gameRunner.run();

	}
}
```

MarioGame.java

```java
package com.in28minutes.learn_spring_framework.game;

public class MarioGame {
	public void up() {
		System.out.println("Jump");
	}

	public void down() {
		System.out.println("Go into a hole");
	}

	public void left() {
		System.out.println("Go back");
	}

	public void right() {
		System.out.println("Accelerate");
	}
}
```

GameRunner.java

```java
/**
 *
 */
package com.in28minutes.learn_spring_framework.game;

/**
 *
 */
public class GameRunner {
	MarioGame game;

	public GameRunner(MarioGame game) {

		this.game = game;
	}

	public void run() {
		System.out.println("Running game: " + game);
		game.up();
		game.down();
		game.left();
		game.right();
	}
}
```

Output:

```
Running game: com.in28minutes.learn_spring_framework.game.MarioGame@279f2327
Jump
Go into a hole
Go back
Accelerate
```

Step-by-Step Summary: Introduction to var in Java

1. **What Did We Do in the Previous Step?**
   - Instead of explicitly declaring variable types, we used `var`:
     - `// MarioGame marioGame = new MarioGame();`
       `var marioGame = new MarioGame();`
     - `// GameRunner gameRunner = new GameRunner(marioGame);`
       `var gameRunner = new GameRunner(marioGame);`
   - This made the code simpler and cleaner.
2. **What is `var`?**
   - `var` is a feature introduced in **Java 10**.
   - It allows you to declare variables without explicitly stating their type.
   - **Type Inference**: The compiler automatically infers the type based on the value assigned.
3. **How Does `var` Work?**
   - Example:
     - `var marioGame = new MarioGame();`
       The compiler sees you are assigning a new `MarioGame` object, so it knows `marioGame` is of type `MarioGame`.
     - `var gameRunner = new GameRunner(marioGame);`
       Similarly, the compiler knows `gameRunner` is of type `GameRunner`.

4. **Benefits of Using var:**
   - **Simplifies Code:**
     You write less code because you don't need to repeat the type.
   - **Improves Readability:**
     Especially useful with complex types, like generics (e.g., List<Map<String, Integer>>).
   - **Reduces Boilerplate:**
     Less repetitive code, making it easier to read and maintain.
5. **Important Notes:**
   - You must use at least **Java 10** to use var.
   - The type is always determined by the value on the right side of the assignment.
6. **Conclusion:**
   - var is a helpful tool for modern Java development.
   - It's especially useful for simplifying code that deals with complex types.
   - You'll continue to see and use var in later parts of the course.

In short:
The var keyword in Java lets you declare variables without specifying their type—the compiler figures it out for you. This makes your code simpler and more readable, especially when dealing with complex types. You need Java 10 or above to use it.

Understanding Tight Coupling in Your Code

1. What Has Changed in Your Code?

- In your AppGamingBasic class, you commented out the Mario game and are now using the Super Contra game:

```
// var marioGame = new MarioGame();
var superContraGame = new SuperContraGame();
var gameRunner = new GameRunner(superContraGame);
gameRunner.run();
```

- You created a new SuperContraGame class, similar to MarioGame, but with its own actions.
- In the GameRunner class, you changed the type from MarioGame to SuperContraGame:

```
private SuperContraGame game;

public GameRunner(SuperContraGame game) {
    this.game = game;
}
```

2. What is Tight Coupling?

- **Tight coupling** means that one class (here, GameRunner) is directly dependent on a specific implementation (SuperContraGame).
- Your current GameRunner can only work with SuperContraGame. If you want to switch back to MarioGame or use any other game, you must change the code

in GameRunner itself (change the type and constructor).

3. Why is Tight Coupling a Problem?

- **Lack of Flexibility:**
  If you want to support multiple games, you have to keep editing the GameRunner class every time. For example, to support MarioGame again, you'd need to change all references in GameRunner from SuperContraGame to MarioGame.
- **More Code Changes:**
  Every time you add a new game (e.g., PacManGame), you need to update GameRunner to accept this new type. This leads to duplicated code and more places where bugs can be introduced.
- **Harder Maintenance:**
  As your codebase grows, tightly coupled code becomes harder to maintain, test, and extend.

4. Real-World Analogy

- Think of a car:
  - The engine is tightly coupled to the car — you can't easily replace it with a different engine.
  - The wheels are loosely coupled — you can easily swap them out for another set.
- In software, we want parts (like GameRunner and the games) to be loosely coupled, so we can easily swap or add new games without changing the runner.

5. Your Output

```
Running game: com.in28minutes.learn_spring_framework.game.SuperContraGame@
279f2327
up
Sit down
Go back
Shoot a bullet
```

- The code works, but only because GameRunner was rewritten to work specifically with SuperContraGame.

6. Key Takeaway

- **Current Limitation:**
  Your code is tightly coupled. If you want to run a different game, you must edit the GameRunner code.
- **Best Practice:**
  You should design GameRunner so that it can work with any game, without being rewritten each time. This is called **loose coupling**.

7. What's Next?

- In the next steps, you'll learn how to use a **Java interface** (e.g., GamingConsole) to allow GameRunner to work with any game that implements that interface.
- This will make your code **loosely coupled**, flexible, and much easier to maintain and extend.

Summary Table

Class	What it does	Problem in current design

MarioGame	Implements Mario actions	Not usable by GameRunner without changes
SuperContraGame	Implements Super Contra actions	GameRunner must be rewritten for this
GameRunner	Runs a specific game	Only works with one game at a time

In short:
Your current design works, but it is tightly coupled—GameRunner can only run the game it was written for. If you want flexibility, you need to make your code loosely coupled using interfaces, which you'll learn next!

AppGamingBasic.java
```
package com.in28minutes.learn_spring_framework;

import com.in28minutes.learn_spring_framework.game.GameRunner;
import com.in28minutes.learn_spring_framework.game.MarioGame;
import com.in28minutes.learn_spring_framework.game.SuperContraGame;

public class AppGamingBasic {

	public static void main(String[] args) {
		// var marioGame = new MarioGame();
		var superContraGame = new SuperContraGame();
		var gameRunner = new GameRunner(superContraGame);
		gameRunner.run();

	}
}
```

MarioGame.java
```
package com.in28minutes.learn_spring_framework.game;

public class MarioGame {
	public void up() {
		System.out.println("Jump");
	}

	public void down() {
		System.out.println("Go into a hole");
	}

	public void left() {
		System.out.println("Go back");
	}

	public void right() {
		System.out.println("Accelerate");
	}
}
```

GameRunner.java
```
/**
 *
 */
package com.in28minutes.learn_spring_framework.game;

/**
 *
 */
```

```
public class GameRunner {
	private SuperContraGame game;

	public GameRunner(SuperContraGame game) {

		this.game = game;
	}

	public void run() {
		System.out.println("Running game: " + game);
		game.up();
		game.down();
		game.left();
		game.right();
	}
}
```

SuperContraGame.java

```
package com.in28minutes.learn_spring_framework.game;

public class SuperContraGame {
	public void up() {
		System.out.println("up");
	}

	public void down() {
		System.out.println("Sit down");
	}

	public void left() {
		System.out.println("Go back");
	}

	public void right() {
		System.out.println("Shoot a bullet");
	}
}
```

Output:

```
Running game: com.in28minutes.learn_spring_framework.game.SuperContraGame@279f2327
up
Sit down
Go back
Shoot a bullet
```

Getting Started with Java Spring Framework - Loose Coupling with Java and Spring

20 June 2025 23:22

Iteration Two: Introducing Interfaces for Loose Coupling

1. The Goal

- Move from *tightly coupled* code to *loosely coupled* code.
- Use a **GamingConsole interface** so that GameRunner can work with any game, not just one specific class.

2. Creating the Interface

- Notice that both MarioGame and SuperContraGame have the same methods: up(), down(), left(), and right().
- Create a new interface called GamingConsole in the same package as the game classes.

```
package com.in28minutes.learn_spring_framework.game;

public interface GamingConsole {
	void up();
	void down();
	void left();
	void right();
}
```

- An **interface** represents a set of common actions multiple classes can perform.

3. Implementing the Interface

- Have SuperContraGame implement the GamingConsole interface:

```
package com.in28minutes.learn_spring_framework.game;

public class SuperContraGame implements GamingConsole{
	public void up() {
		System.out.println("up");
	}

	public void down() {
		System.out.println("Sit down");
	}

	public void left() {
		System.out.println("Go back");
	}

	public void right() {
		System.out.println("Shoot a bullet");
	}
}
```

- No errors will appear because SuperContraGame already has all the required methods.

4. Why is This Useful?

- Now, in your main class (AppGamingBasic), you can use a GamingConsole variable:

```
package com.in28minutes.learn_spring_framework;
```

```
import com.in28minutes.learn_spring_framework.game.GameRunner;
import com.in28minutes.learn_spring_framework.game.MarioGame;
import com.in28minutes.learn_spring_framework.game.SuperContraGame;

public class AppGamingBasic {

	public static void main(String[] args) {
//		var game = new MarioGame();
		var game = new SuperContraGame();
		var gameRunner = new GameRunner(game);
		gameRunner.run();

	}
}
```

- You can run the program, and it works as expected.

5. Switching Games Easily

- If you want to run MarioGame, just change the assignment:
  `// var game = new MarioGame();`
  `var game = new SuperContraGame();`
- But if you try this, you might get a compilation error:
  The constructor GameRunner(MarioGame) is undefined.
- **Why?** Because the GameRunner class still expects a SuperContraGame, not a GamingConsole.

6. Making GameRunner Use the Interface

- Change GameRunner so it uses GamingConsole instead of a specific class:

```
/**
 *
 */
package com.in28minutes.learn_spring_framework.game;

/**
 *
 */
public class GameRunner {
	private GamingConsole game;

	public GameRunner(GamingConsole game) {

		this.game = game;
	}

	public void run() {
		System.out.println("Running game: " + game);
		game.up();
		game.down();
		game.left();
		game.right();
	}
}
```

- All method calls (up(), down(), left(), right()) still work because they are part of the interface.

7. What's the Advantage?

- Now, **no matter which game you want to run**, you do not need to change the GameRunner code.
- Just change which game object you create in AppGamingBasic and pass it to GameRunner.
- The code is now **loosely coupled**:
  - GameRunner depends only on the GamingConsole interface, not on any specific game

implementation.

8. One More Step: MarioGame Implements GamingConsole

- Make sure MarioGame also implements the interface:

```
package com.in28minutes.learn_spring_framework.game;

public class MarioGame implements GamingConsole{
	public void up() {
		System.out.println("Jump");
	}

	public void down() {
		System.out.println("Go into a hole");
	}

	public void left() {
		System.out.println("Go back");
	}

	public void right() {
		System.out.println("Accelerate");
	}
}
```

- Now you can switch between games in your main class **without any compilation errors**:

```
package com.in28minutes.learn_spring_framework;

import com.in28minutes.learn_spring_framework.game.GameRunner;
import com.in28minutes.learn_spring_framework.game.MarioGame;
import com.in28minutes.learn_spring_framework.game.SuperContraGame;

public class AppGamingBasic {

	public static void main(String[] args) {
//		var game = new MarioGame();
		var game = new SuperContraGame();
		var gameRunner = new GameRunner(game);
		gameRunner.run();

	}
}
```

9. Key Observation

- **Before:**
  - GameRunner was tightly coupled to one specific game class.
  - To run a different game, you had to change the GameRunner code.
- **After:**
  - GameRunner is coupled only to the GamingConsole interface.
  - To run a different game, just change the object you pass to GameRunner—no changes in GameRunner itself!

10. What You Achieved

- You introduced a GamingConsole interface.
- Both MarioGame and SuperContraGame implement this interface.
- GameRunner now works with the interface, not with a specific game class.
- **Result:** Your code is now **loosely coupled** and much more flexible and maintainable.

In summary:

By introducing the GamingConsole interface and making your game classes implement it, you decoupled GameRunner from specific games. Now, you can switch games easily without editing GameRunner, achieving loose coupling—a best practice in software design!

Detailed Review: Moving from Tight Coupling to Loose Coupling Using Interfaces

1. What Did We Start With?

- You began with two classes: MarioGame and SuperContraGame.
- Each game class had four methods: up(), down(), left(), and right().
- There was **no interface** at the beginning.

2. The Initial Problem: Tight Coupling

- In the first version, your GameRunner class was tightly coupled to the specific game class you wanted to run.
  - Example:
    ```
    var game = new MarioGame();
    var gameRunner = new GameRunner(game);
    gameRunner.run();
    ```
- If you wanted to run SuperContraGame instead, you had to **change the GameRunner class** to accept SuperContraGame instead of MarioGame.
- This meant **every time you wanted to switch games, you had to edit the GameRunner code.**
- This design is called **tight coupling**: the GameRunner is tightly bound to the specific game it is running.

3. The Solution: Introducing an Interface

- To fix this, you introduced a **GamingConsole interface**.
  - The interface defined four methods: up(), down(), left(), and right().
  - Both MarioGame and SuperContraGame were updated to **implement the GamingConsole interface**.
- The GameRunner class was then updated to depend on the interface (GamingConsole) instead of a concrete game class.

4. How Does the Code Look Now?

- **GamingConsole interface:**
  ```
  public interface GamingConsole {
      void up();
      void down();
      void left();
      void right();
  }
  ```
- **MarioGame** and **SuperContraGame** both implement the interface.
- **GameRunner** takes a GamingConsole as a constructor argument, not a specific game class.
- **Switching games is easy:**
  In your main app class, you can simply swap which game you instantiate:
  ```
  // var game = new MarioGame();
  var game = new SuperContraGame();
  var gameRunner = new GameRunner(game);
  gameRunner.run();
  ```
- **No changes are needed in GameRunner** when you want to run a different game.

5. The Benefit: Loose Coupling

- **Flexibility:**
  You can change the game being run by just changing one line in your main class.
- **No need to modify GameRunner:**
  The runner class never needs to know about the specific game class—it only knows about the interface.
- **Result:**
  The code is now **loosely coupled**. Changing the functionality (the game) does not require changes in the code that runs the functionality (the runner).

6. Why Are Interfaces Powerful?

- Interfaces are a core concept in Java.
- They allow you to define a contract (set of methods) that multiple classes can implement in their own way.
- Mastering interfaces is crucial for becoming a strong Java developer.

7. What's Next? Exercise Time!

- To prove your code is now really loosely coupled, you're given an exercise:
  - **Create a new class called PacmanGame.**
  - Make PacmanGame implement the GamingConsole interface.
  - Try this yourself before moving to the next step, where the instructor will provide a solution.

In summary:
You started with tightly coupled code, improved it by introducing a GamingConsole interface, and made your application loosely coupled and flexible. Now, you're ready to add even more games (like Pacman) without ever changing the GameRunner class again!

Review & Solution: Creating PacmanGame with Loose Coupling

1. Exercise Recap

- The exercise was to create a new game class called PacmanGame.
- PacmanGame should implement the GamingConsole interface, just like MarioGame and SuperContraGame.

2. Solution Steps

a. Copy and Adapt Existing Code

- The easiest way to start was to copy the code from MarioGame:
  - Select the entire MarioGame class (Ctrl+C), paste it into a new file (Ctrl+V).
  - Rename the new class to PacmanGame.
  - Save all files to make sure your changes are applied.

b. Update the Class Name

- Rename MarioGame to PacmanGame in the new file.
- Now you have a PacmanGame class in your project.

c. Implement the Interface

- Ensure PacmanGame implements GamingConsole.
- Define the four methods: up(), down(), left(), and right().
- For simplicity, you can just print "up", "down", "left", and "right" in these methods.

```
package com.in28minutes.learn_spring_framework.game;

public class PacmanGame implements GamingConsole{
	public void up() {
		System.out.println("up");
	}

	public void down() {
		System.out.println("down");
	}

	public void left() {
		System.out.println("left");
	}

	public void right() {
		System.out.println("right");
	}
}
```

d. Use PacmanGame in the Main Application

- In your AppGamingBasic (main) class:
  - Change the game assignment to:

```
package com.in28minutes.learn_spring_framework;

import com.in28minutes.learn_spring_framework.game.GameRunner;
import com.in28minutes.learn_spring_framework.game.MarioGame;
import com.in28minutes.learn_spring_framework.game.PacmanGame;
import com.in28minutes.learn_spring_framework.game.SuperContraGame;

public class AppGamingBasic {

	public static void main(String[] args) {
//		var game = new MarioGame();
//		var game = new SuperContraGame();
		var game = new PacmanGame();
		var gameRunner = new GameRunner(game);
		gameRunner.run();

	}
}
```

  If you see a compilation error, organize your imports to resolve it.

e. Run the Application

- Right-click the main class and select **Run As > Java Application**.
- You should see output indicating that you are running the Pacman game, showing the actions: up, down, left, right.

3. Key Learning: Loose Coupling with Interfaces

- Thanks to the interface, you didn't need to make any changes to GameRunner to run PacmanGame.
- GameRunner now interacts with any game class that implements GamingConsole.
- You can swap out which game you run by simply changing one line in your main class.
- This demonstrates the power of loose coupling: your runner is not tied to any specific implementation.

4. End of Iteration Two

- You have now completed iteration two, where you achieved loose coupling using interfaces.
- Your code is flexible and maintainable: to add a new game, just implement the interface—no need to touch the runner logic.

5. Next Steps

- In the next phase, you will learn how to bring in the **magic of the Spring Framework** to make your application even more powerful and flexible.

In summary:
You created a PacmanGame class by copying and adapting the MarioGame class, made it implement the GamingConsole interface, and verified that you could run it without changing the GameRunner. This proves your code is now loosely coupled, thanks to interfaces. Next, you'll integrate Spring to further enhance your app!

Current State of the Code: Manual Object Creation and Dependency Wiring

1. What Does the Code Look Like Now?

- You are creating a PacmanGame object.
- You are also creating a GameRunner object.
- You pass the PacmanGame (or any game) as an argument to the GameRunner constructor.
- Example:

```
package com.in28minutes.learn_spring_framework;

import com.in28minutes.learn_spring_framework.game.GameRunner;
import com.in28minutes.learn_spring_framework.game.MarioGame;
import com.in28minutes.learn_spring_framework.game.PacmanGame;
import com.in28minutes.learn_spring_framework.game.SuperContraGame;

public class AppGamingBasic {

	public static void main(String[] args) {
//		var game = new MarioGame();
//		var game = new SuperContraGame();
		var game = new PacmanGame(); // 1: Object Creation
		var gameRunner = new GameRunner(game);
		// 2: Object Creation + Wiring of Dependencies
		// Game is Dependency of GameRunner
		gameRunner.run();

	}
}
```

2. What is Actually Happening Here?

- **Object Creation:**
  - You create a game object (e.g., PacmanGame).
  - You create a game runner object (GameRunner).
- **Wiring of Dependencies:**
  - The GameRunner class needs a game to run, so the game object is a **dependency** of GameRunner.
  - When you pass the game to the GameRunner constructor, you are **injecting** or **wiring** this dependency.
  - This means the game instance is provided to the GameRunner so it can function.

3. What Does "Wiring Dependencies" Mean?

- **Dependency:**
  - If a class needs another object to work, that object is called a dependency.
  - In this case, a game (like MarioGame, SuperContraGame, or any GamingConsole) is a dependency of GameRunner.
- **Dependency Injection:**
  - The process of providing an object's dependencies from the outside (instead of the object creating them itself) is called dependency injection.
  - Here, you create the game and pass it into the GameRunner — this is *manual dependency injection*.

4. Why Is This Important in Real Applications?

- In enterprise (large-scale) applications:
  - You often have **thousands of classes**.
  - There are **thousands of dependencies** that need to be created and injected in the right places.
  - Managing all this manually would be very complex and error-prone.

5. The Current Limitation

- Right now, **you** are responsible for:
  - Creating all the objects (game, game runner, etc.).
  - Wiring them together (passing dependencies to constructors).
- All of this is managed in your code, inside the Java Virtual Machine (JVM).

6. What's the Next Step? Bring in the Spring Framework

- The goal is to **let the Spring Framework manage object creation and wiring** for you.
- Spring can:
  - Create objects automatically.
  - Inject (wire) dependencies wherever they are needed.
  - Manage the lifecycle of these objects (called "beans" in Spring).

7. How Will You Start with Spring?

- Before jumping into using Spring with the GameRunner example, you will:
  - Start with a much **simpler example**.
  - Use Spring to manage basic objects (like name, age, Person, and their Address).
  - See how Spring creates and wires these objects for you automatically.

8. What's Next?

- Get ready to let the Spring Framework manage your objects and their dependencies!
- In the next step, you'll begin with simple examples to see Spring in action.

In summary:
Up to now, you have been creating and wiring objects and their dependencies manually in your code. In large applications, this quickly becomes difficult to manage. The next step is to let the Spring Framework take over this responsibility, starting with simple object management before applying it to more complex scenarios like your game runner.

Step-by-Step: Creating Your First Spring Bean and Spring Context

1. Refactor and Set Up Your Classes

- You started with a class called App**GamingBasicJava**.
- You refactored (renamed) this class to App01**GamingBasicJava** for better organization.
  - Right-clicked, chose Refactor > Rename, and entered App01**GamingBasicJava**.
- You deleted the default **LearnSpringFrameworkApplication.java** class that was auto-generated when you created the Spring project, since you no longer needed it.
  - Right-clicked on the class and deleted it.
- Now, you have a single entry-point Java application: App01**GamingBasicJava**.
- Running this class (with Pacman) still works, and you see Pacman's actions printed in the console.

2. Prepare for Your First Spring Example

- You copied App01**GamingBasicJava** (Ctrl+C, Ctrl+V) to create a new class.
- Renamed this new class to App02**HelloWorldSpring**.
- Opened App02**HelloWorldSpring** and cleared out all the existing code.
- Saved and organized imports, so you have a clean slate to start learning Spring.

3. The Goal: Let Spring Manage a Simple Object

- You want to create and manage a simple string bean (like "name") using the Spring Framework.
- Rather than creating and wiring objects yourself, you want Spring to do this for you.
- To do this, you need to:
  1. **Launch a Spring Application Context** (Spring's container inside the JVM).
  2. **Tell Spring what to manage** (which beans/objects you want it to create).

4. How Do You Configure Spring?

- One common way is to use a **configuration class**.
  - This is a regular Java class annotated with @Configuration.
  - In this class, you will define beans (objects) that Spring should manage.

5. Create a Configuration Class

- In your main package, right-click and select New > Java Class.
- Name this class **HelloWorldConfiguration**.
- In **HelloWorldConfiguration.java**, annotate the class with @Configuration.
  - This tells Spring that this class will provide bean definitions.
  - Organize imports, which brings in org.springframework.context.annotation.Configuration.

```
package com.in28minutes.learn_spring_framework;

import org.springframework.context.annotation.Configuration;

@Configuration
public class HelloWorldConfiguration {

}
```

What does @Configuration do?

- As per the Javadoc, it indicates that the class declares one or more bean methods and can be processed by the Spring container to generate bean definitions and service requests.

6. Launch a Spring Context

- In App02**HelloWorldSpring**, you want to launch a Spring application context using your configuration class.
- You use the **AnnotationConfigApplicationContext** class for this:
  `var context = new AnnotationConfigApplicationContext(HelloWorldConfiguration.class);`
- This creates a Spring context (container) inside your JVM, using your configuration class.

```
package com.in28minutes.learn_spring_framework;
```

```java
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class App02HelloWorldSpring {

	public static void main(String[] args) {
		// 1: Launch a Spring Context

		var context = new
AnnotationConfigApplicationContext(HelloWorldConfiguration.class);
		// 2: Configure the things that we want spring to manage - @Configuration
	}
}
```

7. Running the Application

- Run your App02HelloWorldSpring Java application.
- You see that there are **no errors**.
  - Although nothing prints out yet, Spring is doing a lot behind the scenes to set up the context and prepare for bean management.

8. What Have You Achieved?

- You have successfully launched a Spring context from your Java application using a configuration class.
- You are now ready to let Spring manage your beans (objects).

9. Quick Review

- Whenever you run a Java application, it runs inside the **JVM**.
- Inside the JVM, you now have a **Spring Application Context** running.
- You created this context using the AnnotationConfigApplicationContext and your HelloWorldConfiguration class.
- This sets the stage for Spring to create and manage objects for you.

10. What's Next?

- This might seem complex at first, but you will review and deepen your understanding in the next step.
- Soon, you will actually add beans to the configuration and see Spring manage them for you.

In summary:
You refactored your classes, created a new clean app for your Spring experiments, wrote a configuration class with @Configuration, and launched a Spring context using AnnotationConfigApplicationContext. You are now ready to see Spring manage your first real bean!

Step-by-Step: Your First Spring Bean

1. Reviewing What You Did So Far

- You created a configuration class:
  @Configuration
  public class HelloWorldConfiguration { }
  - The @Configuration annotation tells Spring that this class contains bean definitions to be managed by the Spring container.
- In your main application class (App02HelloWorldSpring), you launched a Spring context:
  var context = **new** AnnotationConfigApplicationContext(HelloWorldConfiguration.**class**);
  - This creates a Spring context (or container) inside the JVM, using your configuration class.

2. Why This Matters

- The configuration class by itself is not enough; you need to actually launch it using the Spring context.
- The line above initializes the Spring container, which manages the beans you will define.

3. Adding Your First Bean

- You want Spring to manage a simple String object named "name".
- To do this, go to HelloWorldConfiguration and add:

```
@Bean
public String name() {
    return "Nilesh";
}
```

  - The @Bean annotation tells Spring that the return value of this method should be managed as a bean.
  - The bean's name defaults to the method name (name).
- Now, your configuration class looks like:

```
@Configuration
public class HelloWorldConfiguration {
    @Bean
    public String name() {
        return "Nilesh";
    }
}
```

4. Running Your Application

- Run App02HelloWorldSpring as a Java application.
- There are no errors, which means your Spring context is up and running and managing your bean.

5. Retrieving the Bean

- To see the bean in action, get it from the context:

```
System.out.println(context.getBean("name"));
```

  - context.getBean("name") retrieves the bean by its name (which matches your method name).
  - You print it to the console, so you should see Ranga output.

6. What Happened Here?

- You launched a JVM (by running your Java application).
- Inside the JVM, you started a Spring Application Context using your configuration class.
- You defined a bean (name) in the configuration class using @Bean.
- Spring manages this bean for you.
- You retrieved the bean using context.getBean("name") and printed it.

7. Key Concepts Introduced

- **Spring Context / Container:** The environment where Spring manages your beans.
- **Configuration Class:** Where you tell Spring what to manage, using @Configuration.
- **Bean:** Any object managed by Spring (defined with @Bean).
- **Dependency Injection:** Spring can provide these beans to other parts of your application as needed.
- **Retrieving Beans:** Use context.getBean("beanName") to get your beans.

8. What's Next?

- You may have questions about how Spring works and why certain things are done this way.
- In the next steps, you will explore more ways of defining and retrieving beans, and dig deeper into Spring's features.

In summary:
You created a configuration class, defined your first Spring bean, launched a Spring context, and retrieved your bean from the Spring container. You saw how Spring can manage objects for you, setting the foundation for more advanced dependency injection and bean management.

Code:

App02HelloWorldSpring.java

```java
package com.in28minutes.learn_spring_framework;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class App02HelloWorldSpring {

	public static void main(String[] args) {
		// 1: Launch a Spring Context

		var context = new AnnotationConfigApplicationContext(HelloWorldConfiguration.class);

		// 2: Configure the things that we want spring to manage -
		// HelloWorldConfiguration - @Configuration
		// name - @Bean

		// 3: Retrieving Beans managed by Spring
		System.out.println(context.getBean("name"));

	}
}
```

HelloWorldConfiguration.java

```java
package com.in28minutes.learn_spring_framework;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class HelloWorldConfiguration {

	@Bean
	public String name() {
		return "Nilesh";
	}
}
```

Quick Code Review: Your First Spring Bean

1. What Have You Achieved?

- You successfully got Spring to manage a simple bean inside the JVM.
- You now have a Spring context running inside your Java application.

2. Reviewing the Code

a. Configuration Class

- You defined a configuration class using the @Configuration annotation:

```
@Configuration
public class HelloWorldConfiguration {
    @Bean
    public String name() {
        return "Nilesh";
    }
}
```

- This is where you define all the beans you want Spring to manage.
- The @Bean annotation on the name() method tells Spring to create and manage a bean named "name" with the value "Nilesh".

b. Launching the Spring Context

- In your main application class:
  `var context = new AnnotationConfigApplicationContext(HelloWorldConfiguration.class);`
- This line launches the Spring context using your configuration class.
- At this point, the Spring context is ready, and all beans defined in your configuration are created and managed by Spring.

c. Retrieving a Bean

- You retrieve the bean using its name:
  `System.out.println(context.getBean("name"));`
- The string "name" matches the method name in your configuration class.
- This prints the value of the bean, which is "Nilesh".

3. Key Concepts Highlighted

- **@Configuration:** Marks a class as a source of bean definitions for the Spring container.
- **@Bean:** Marks a method as a bean producer; the method name is used as the bean's name.
- **Spring Context (Container):** The environment where Spring creates, manages, and wires your beans.
- **context.getBean("name"):** Retrieves the bean by its name from the Spring context.

4. Summary of the Flow

1. **Define a configuration class** and mark it with @Configuration.
2. **Create beans** inside this class using @Bean.
3. **Launch the Spring context** with AnnotationConfigApplicationContext, passing your configuration class.
4. **Retrieve and use beans** from the context with context.getBean("beanName").

5. Congratulations & Next Steps

- **Congratulations** on creating your first Spring bean!
- You now have the foundation to create and manage more beans using Spring.
- In the upcoming steps, you'll create additional beans and experiment further to understand Spring's capabilities.

In summary:
You defined a configuration class, created a simple bean with @Bean, launched a Spring context, and successfully retrieved your bean by name. You're now ready to explore more powerful features of Spring in the next steps!

Getting Started with Java Spring Framework - 4 - Playing with Spring Framework

25 June 2025 12:45

Step-by-Step: Creating and Managing More Beans with Spring

1. Creating More Beans in the Configuration Class

- You go back to your HelloWorldConfiguration class to add more beans for Spring to manage.

a. Add an Integer Bean

- You define a new bean for age:
```
@Bean
public int age() {
    return 15;
}
```
- This means Spring will manage an age bean with the value 15.
- In your main method, you can retrieve and print it:
```
System.out.println(context.getBean("age"));
```
- Running the application will print 15.

2. Creating and Managing Beans of Custom Classes

a. Introduce Java Records

- You create a record called Person:
```
public record Person(String name, int age) {}
```
- **What is a record?**
  - Records are a Java feature (introduced in JDK 16) to reduce boilerplate code.
  - With records, you don't need to write getters, constructors, equals(), hashCode(), or toString() — they are generated automatically.

b. Define a Person Bean

- Add a bean method in your configuration:
```
@Bean
public Person person() {
    return new Person("Ravi", 20);
}
```
- Now, Spring manages a person bean.
- In your main class, retrieve and print it:
```
System.out.println(context.getBean("person"));
```
- Output will look like: Person[name=Ravi, age=20]
  - The record's toString() is automatically provided.

3. Exercise: Create an Address Record and Bean

a. Define the Record

- Create a record called Address:
```
public record Address(String firstLine, String city) {}
```
  - (Remember to add {} at the end, even if it's empty.)

b. Define an Address Bean

- In your configuration, add:

```
@Bean
public Address address() {
    return new Address("Baker Street", "London");
}
```

- Now, Spring manages an address bean.

c. Retrieve and Print the Address Bean

- In your main class:
  `System.out.println(context.getBean("address"));`
- Output will look like: Address[firstLine=Baker Street, city=London]

4. Key Concepts Reinforced

- **Beans can be of any type:** primitives, Strings, or custom classes like records.
- **Spring manages the lifecycle and retrieval** of all these beans.
- **Records** make it easy to create immutable data structures for use as beans.
- **Retrieving beans:** You use context.getBean("beanName") to get and use any bean managed by Spring.

5. Troubleshooting

- If you see a compilation error when defining a record, check the syntax: you need parentheses for the parameters and curly braces {} at the end, even if empty.
- Example: public record Address(String firstLine, String city) {}

6. Summary of This Step

- You added more beans to your configuration: a primitive (int age), a Person record, and an Address record.
- You learned how to retrieve and print these beans from the Spring context.
- You saw how records simplify creating beans for Spring to manage.
- This step may seem basic, but it's the foundation for more advanced Spring features you'll explore next.

In summary:
You expanded your Spring configuration to manage several beans, including primitives and custom record types. You learned how to define, retrieve, and print these beans, setting up the groundwork for more complex Spring applications.

Code:
App02HelloWorldSpring.java

```
package com.in28minutes.learn_spring_framework;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class App02HelloWorldSpring {

	public static void main(String[] args) {
		// 1: Launch a Spring Context

		var context = new AnnotationConfigApplicationContext(HelloWorldConfiguration.class);

		// 2: Configure the things that we want spring to manage -
		// HelloWorldConfiguration - @Configuration
		// name - @Bean

		// 3: Retrieving Beans managed by Spring
```

```
            System.out.println(context.getBean("name"));

            System.out.println(context.getBean("age"));

            System.out.println(context.getBean("person"));

            System.out.println(context.getBean("address"));
        }
}
```

HelloWorldConfiguration.java

```
package com.in28minutes.learn_spring_framework;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;


// Eliminate verbosity in creating Java Beans
// Public accessor methods, constructor,
// equals , hashcode and toString are automatically created.
// Released in JDK 16.

record Person (String name, int age) {};


// Address - firstLine & city
record Address(String firstLine, String city) {};

@Configuration
public class HelloWorldConfiguration {

        @Bean
        public String name() {
                return "Nilesh";
        }

        @Bean
        public int age() {
                return 15;
        }

        @Bean
        public Person person() {
                return new Person("Ravi", 20);
        }

        @Bean
        public Address address() {
                return new Address("Baker Street", "London");
        }

}
```

Step-by-Step: Bean Naming, Retrieving, and Using Beans as Dependencies

1. Review: Managing Multiple Beans

- You now have Spring managing several beans:

    - name (String)
    - age (int)
    - person (record)
    - address (record)
- By default, **the bean name is the method name** in your @Configuration class.

2. Customizing Bean Names

- If you want to customize the bean's name, use the @Bean(name = "customName") attribute.
```
@Bean(name = "address2")
public Address address() {
    return new Address("Baker Street", "London");
}
```
- Now, Spring registers this bean as address2 instead of address.

3. What Happens If You Use the Old Name?

- If you try to retrieve a bean using the old name (address), you get an exception:
  No bean named 'address' available
- If you use the new name (address2), it works:
```
System.out.println(context.getBean("address2"));
```

4. Retrieving Beans by Type

- You don't always have to use the bean name.
- You can also retrieve beans by their type:
```
System.out.println(context.getBean(Address.class));
```
- If only one bean of that type exists, this works perfectly.
- Output is the same as retrieving by name.

5. Key Takeaways on Bean Retrieval

- **Retrieve by name:**
  context.getBean("beanName")
- **Retrieve by type:**
  context.getBean(BeanType.class)
- Both approaches are supported by Spring, giving you flexibility.

6. Next: Using Existing Beans as Dependencies

- You've created beans with direct values so far.
- Now, you will learn how to create a new bean (e.g., person2) **using other beans already managed by Spring** (like name and age).
- This is a key feature of Spring: it can manage dependencies between your beans automatically.
- You will see how to **inject** existing beans into new beans in the next step.

In summary:
You learned how to customize bean names with @Bean(name="..."), how to retrieve beans by name and by type, and saw that Spring provides multiple ways to access your managed objects. Next, you'll see how to wire beans together, letting Spring inject dependencies for

Code:

App02HelloWorldSpring.java
```
package com.in28minutes.learn_spring_framework;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;
```

```
public class App02HelloWorldSpring {

	public static void main(String[] args) {
		// 1: Launch a Spring Context

		var context = new AnnotationConfigApplicationContext(HelloWorldConfiguration.class);

		// 2: Configure the things that we want spring to manage -
		// HelloWorldConfiguration - @Configuration
		// name - @Bean

		// 3: Retrieving Beans managed by Spring
		System.out.println(context.getBean("name"));

		System.out.println(context.getBean("age"));

		System.out.println(context.getBean("person"));

		System.out.println(context.getBean("address2"));

		System.out.println(context.getBean(Address.class));
	}
}
```

HelloWorldConfiguration.java

```
package com.in28minutes.learn_spring_framework;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;


// Eliminate verbosity in creating Java Beans
// Public accessor methods, constructor,
// equals , hashcode and toString are automatically created.
// Released in JDK 16.

record Person (String name, int age) {};


// Address - firstLine & city
record Address(String firstLine, String city) {};

@Configuration
public class HelloWorldConfiguration {

	@Bean
	public String name() {
		return "Nilesh";
	}

	@Bean
	public int age() {
		return 15;
	}

	@Bean
	public Person person() {
		return new Person("Ravi", 20);
	}

	@Bean(name = "address2")
	public Address address() {
		return new Address("Baker Street", "London");
```

```
        }

}

Output:
Nilesh
15
Person[name=Ravi, age=20]
Address[firstLine=Baker Street, city=London]
Address[firstLine=Baker Street, city=London]
```

Step-by-Step: Creating Beans Using Other Beans & Advanced Bean Management

1. Objective

- **Create a new bean (person2 and person3) using values from other beans managed by Spring** (like name, age, and address).
- **Understand two ways to do this:**
  - By method calls
  - By method parameters

2. Creating person2 Bean by Method Calls

- **Define a new bean method for person2:**

```
@Bean
public Person person2MethodCall() {
    return new Person(name(), age(), address2());
}
```

  - Here, you directly call the other bean methods (name(), age(), address2()) inside the configuration class.
  - This way, Spring ensures that the same singleton bean instance is used each time.
- **Retrieve and print the bean:**

```
System.out.println(context.getBean("person2MethodCall"));
```

- **Result:** Output displays a Person created using the values of the other beans (name, age, and address2).

3. Adding Address to the Person Record

- **Update your Person record** to include an Address:

```
record Person(String name, int age, Address address) {}
```

- **Update the beans accordingly:**
  - For person, you can hardcode the address:

```
@Bean
public Person person() {
    return new Person("Ravi", 20, new Address("Main Street", "Utrecht"));
}
```

  - For person2MethodCall, use the address2() bean (as above).

4. Creating person3 Bean by Method Parameters

- **Define a bean method with parameters:**
  ```
  @Bean
  public Person person3Parameters(String name, int age, Address address3) {
      return new Person(name, age, address3);
  }
  ```
  - Spring will **automatically inject** beans matching the parameter names (name, age, and address3).
  - This is a more declarative and cleaner way to wire dependencies.
- **Create another address bean:**
  ```
  @Bean(name = "address3")
  public Address address3() {
      return new Address("Moti Nagar", "Hyderabad");
  }
  ```
- **Retrieve and print the bean:**
  ```
  System.out.println(context.getBean("person3Parameters"));
  ```
- **Result:** Output displays a Person with the same name and age, but with a new address (Moti Nagar, Hyderabad).

5. Key Concepts and Learnings

- **Custom Bean Names:**
  Use @Bean(name = "customName") to assign a custom name to a bean.
- **Bean Retrieval:**
  - By name: context.getBean("beanName")
  - By type: context.getBean(Address.class)
- **Dependency Injection Between Beans:**
  - **Method call approach:** Call other bean methods directly in your configuration.
  - **Parameter approach:** Use method parameters named after beans; Spring injects them automatically.
- **Multiple Beans of the Same Type:**
  If you define more than one bean of the same type (e.g., two addresses: address2, address3), context.getBean(Address.class) throws an error:
  NoUniqueBeanDefinitionException: expected single matching bean but found 2: address2,address3
  - This is because Spring doesn't know which bean to inject when there's more than one of the same type.

6. Troubleshooting and Output

- If you see an error about multiple beans when calling context.getBean(Address.class), it's because Spring finds more than one bean of that type.
  - **Solution:** Retrieve by name, or use qualifiers (to be covered later).

7. Summary of This Step

- You learned to:
  - Assign custom names to beans.
  - Retrieve beans by name or type.
  - Create new beans using existing beans (dependency injection).
  - Use method parameters for automatic bean injection.
  - Handle issues when multiple beans of the same type exist.
- **This is the foundation for more powerful dependency management in Spring!**

In summary:
You explored custom bean naming, multiple ways to retrieve beans, and how to create beans using other beans (dependency injection)—both by method calls and method parameters. You also learned about

issues that can arise with multiple beans of the same type, and how Spring manages dependencies for you.

Code:

App02HelloWorldSpring.java

```
package com.in28minutes.learn_spring_framework;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class App02HelloWorldSpring {

	public static void main(String[] args) {
		// 1: Launch a Spring Context

		var context = new AnnotationConfigApplicationContext(HelloWorldConfiguration.class);

		// 2: Configure the things that we want spring to manage -
		// HelloWorldConfiguration - @Configuration
		// name - @Bean

		// 3: Retrieving Beans managed by Spring
		System.out.println(context.getBean("name"));

		System.out.println(context.getBean("age"));

		System.out.println(context.getBean("person"));

		System.out.println(context.getBean("person2MethodCall"));

		System.out.println(context.getBean("person3Parameters"));

		System.out.println(context.getBean("address2"));

//		System.out.println(context.getBean(Address.class));
	}
}
```

HelloWorldConfiguration.java

```
package com.in28minutes.learn_spring_framework;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;


// Eliminate verbosity in creating Java Beans
// Public accessor methods, constructor,
// equals , hashcode and toString are automatically created.
// Released in JDK 16.

record Person (String name, int age, Address address) {};


// Address - firstLine & city
record Address(String firstLine, String city) {};

@Configuration
public class HelloWorldConfiguration {

	@Bean
	public String name() {
		return "Nilesh";
	}
```

```java
	@Bean
	public int age() {
		return 15;
	}

	@Bean
	public Person person() {
		return new Person("Ravi", 20, new Address("Miain Street", "Utrecht"));
	}

	@Bean
	public Person person2MethodCall() {
		return new Person(name(), age(), address());
	}

	@Bean
	public Person person3Parameters(String name, int age, Address address3) { // name, age, address2
		return new Person(name, age, address3);
	}

	@Bean(name = "address2")
	public Address address() {
		return new Address("Baker Street", "London");
	}

	@Bean(name = "address3")
	public Address address3() {
		return new Address("Motinagar", "Hyderabad");
	}
}
```

Code Review: Wiring Beans with Method Calls & Parameters

1. Configuration Overview

- You have a configuration class, HelloWorldConfiguration, marked with @Configuration.
- You are defining multiple beans using the @Bean annotation.
- You are using Java records for concise bean definitions (for Person and Address).

2. The Basic Beans

- **name bean:**
  ```java
  @Bean
  public String name() {
     return "Nilesh";
  }
  ```
- **age bean:**
  ```java
  @Bean
  public int age() {
     return 15;
  }
  ```

3. Beans with Dependencies (Wiring Approaches)

A. Wiring Using Method Calls

- **Example: person2MethodCall**

```
@Bean
public Person person2MethodCall() {
    return new Person(name(), age(), address());
}
```

  - Here, you **call other bean methods** (name(), age(), and address()) directly within the bean method.
  - Spring ensures that these calls return the managed singletons, not new instances.

B. Wiring Using Method Parameters

- **Example: person3Parameters**

```
@Bean
public Person person3Parameters(String name, int age, Address address3) {
    return new Person(name, age, address3);
}
```

  - Here, you **declare method parameters** that match the bean names (name, age, and address3).
  - Spring **automatically injects the beans** with matching names/types into this method.
  - This is called *autowiring by parameter*.

4. Address Beans (with Custom Names)

- **address2 bean:**

```
@Bean(name = "address2")
public Address address() {
    return new Address("Baker Street", "London");
}
```

- **address3 bean:**

```
@Bean(name = "address3")
public Address address3() {
    return new Address("Motinagar", "Hyderabad");
}
```

5. Retrieving Beans in Your Main App

- In App02HelloWorldSpring, you retrieve and print each bean:

```
System.out.println(context.getBean("name"));
System.out.println(context.getBean("age"));
System.out.println(context.getBean("person"));
System.out.println(context.getBean("person2MethodCall"));
System.out.println(context.getBean("person3Parameters"));
System.out.println(context.getBean("address2"));
// System.out.println(context.getBean(Address.class)); // commented out: multiple beans of type Address
```

- Output will show the details for each bean, including those created by wiring with method calls and parameters.

6. Key Learnings & Takeaways

- **Method Call Wiring**: Directly calls other bean methods in the config class; Spring returns the managed singleton.
- **Parameter Wiring**: Use method parameters named after beans; Spring injects the right beans by name and type.
- **Custom Bean Names**: Use @Bean(name = "customName") to set a specific name for a bean.
- **Autowiring**: Spring's ability to inject dependencies automatically, making your code more maintainable and testable.
- **Multiple Beans of the Same Type**: If you have more than one bean of a given type (e.g.,

two `Address` beans), retrieving by type with `context.getBean(Address.class)` will fail with a "NoUniqueBeanDefinitionException".

7. Why This Matters

- These two wiring approaches scale to real-world Spring applications with many dependencies.
- Understanding both methods gives you flexibility when designing your bean configuration and dependency management in Spring.

In summary:
You now have a Spring configuration using both method calls and method parameters to wire dependencies. You've seen how to use custom bean names, how Spring injects beans by name/type, and why retrieving by type can fail if there are multiple beans of the same class.

You're well-prepared to build more advanced Spring applications and manage dependencies effectively!

Getting Started with Java Spring Framework - 5 - Key Questions about Spring

26 June 2025 23:22

Spring Beans: Questions & Key Concepts to Explore

1. Spring Container, Context, IOC Container, ApplicationContext

- Throughout the last steps, you've seen terms like **container** and **context** used repeatedly.
- **What do these mean?**
  - **Spring Container**: The core part of the Spring Framework that manages the lifecycle and configuration of application objects.
  - **Spring Context**: A specific type of container that provides more features (like internationalization, event propagation, etc.). Often, "container" and "context" are used interchangeably in Spring.
  - **IOC Container**: Stands for Inversion of Control Container, which means Spring, not you, controls the creation and wiring of objects (beans).
  - **ApplicationContext**: The most commonly used Spring context/container, which you've been using via `AnnotationConfigApplicationContext`.

2. Java Bean vs. Spring Bean

- You've also used the term **bean** often.
  - **Java Bean**: A Java class that follows conventions like having a no-argument constructor, getters/setters, and being serializable.
  - **Spring Bean**: Any object that is managed by the Spring container. It could be any Java object, not just a traditional Java bean.

3. Listing All Beans Managed by Spring

- As you've added more beans, you may wonder:
  - **How do I list all the beans that Spring is managing?**
  - There are ways to query the Spring context and print out all bean names/types.

4. Multiple Matching Beans of the Same Type

- **What happens if you have more than one bean of the same type?**
  - Example: If you have two beans of type `Address` (`address2` and `address3`) and try to retrieve `context.getBean(Address.class)`, you get an error.
  - **Why?** Because Spring finds multiple beans of the same type and doesn't

know which one to pick.
    - **How can you tell Spring to prioritize one?**
        - There are mechanisms (like `@Primary` and `@Qualifier`) to resolve such ambiguities.

5. Who Creates the Objects?

- You might notice: "We're still writing `return new Person(...)` in our configuration! Aren't we still the ones creating the objects?"
    - **Shouldn't Spring be creating the objects for us?**
    - There are ways to let Spring take over the instantiation completely, especially when using Spring's component scanning and dependency injection features.

6. Looking Ahead

- These are all **fundamental questions** that will help deepen your Spring understanding:
    - What is the difference between all the Spring containers and contexts?
    - What's the real distinction between Java beans and Spring beans?
    - How do I get a list of all beans in my Spring context?
    - What if there are multiple beans of the same type—how do I control which is used?
    - How do I let Spring create and inject objects for me, rather than writing all the `new` statements myself?
- **You will get clear answers to all these questions in the upcoming steps.**

In summary:
You've laid a strong foundation in Spring, but naturally, some big questions remain—about containers, beans, autowiring, and bean selection. The next lessons will answer these questions and help you gain real mastery of Spring's core concepts.

Answering: What is a Spring Container?

1. Definition

- A **Spring container** is the core part of the Spring Framework that manages Spring beans and their lifecycle.
- It is responsible for creating, configuring, and managing your application's objects (beans).

2. Inputs & Outputs

- **Inputs to the Spring container:**
    - Your Java classes (including any custom classes and configuration files).

    - A configuration file (e.g., `HelloWorldConfiguration`) that contains bean definitions.
- **Output of the Spring container:**
    - A *ready system* inside your JVM: a Spring context that manages all your configured beans.
    - At runtime, this means your beans are created, wired together, and managed by Spring.

3. Terminology

- **Spring container** is also referred to as:
    - **Spring context**
    - **IOC Container** (IOC = Inversion of Control)
- All these terms refer to the same core concept: the mechanism that takes your configuration and classes, then creates and manages the system for you.

4. The Role of the Container

- The IOC container creates the runtime system.
- It manages the **lifecycle** of beans (creation, initialization, destruction).
- It takes over the responsibility of wiring dependencies and managing object creation, rather than you doing it manually.

5. Types of Spring Containers

There are two popular types of IOC containers in Spring:

A. BeanFactory

- The most basic Spring container.
- Provides the fundamental features of dependency injection.
- Rarely used directly in modern enterprise applications.

B. ApplicationContext

- An advanced Spring container with additional enterprise features.
- Supports internationalization, event propagation, integration with Spring AOP (Aspect-Oriented Programming), and more.
- **Recommended and most frequently used** in enterprise applications, including web apps, REST APIs, and microservices.
- In your code, you are using `AnnotationConfigApplicationContext`, which is a type of `ApplicationContext`.

6. Practical Recommendation

- In most real-world and enterprise applications, you will use `ApplicationContext`.
- `BeanFactory` is only suitable for very specialized use cases (e.g., extremely memory-constrained environments like certain IoT devices).
- The instructor notes that in over 20 years of using Spring, they have never used `BeanFactory` directly.

7. Summary

- **Spring container/Spring context/IOC container**: All mean the same thing—the system that takes your classes and configuration, then creates and manages your beans.

- **ApplicationContext** is the recommended container for almost all scenarios.
- As you continue, you'll explore more features and details of Spring containers.

In summary:
The Spring container (also called Spring context or IOC container) is what manages your beans and their lifecycle, wiring dependencies for you. ApplicationContext is the advanced, most commonly used type of container, while BeanFactory is rarely used directly.

POJO vs Java Bean vs Spring Bean

1. What is a POJO?

- **POJO** stands for **Plain Old Java Object.**
- It is any simple Java class:
  - It can have any variables and methods.
  - There are **no restrictions or requirements** (no need for specific constructors, methods, or interfaces).
  - All classes you write in Java are POJOs by default.
- **Example:**

```
public class MyPojo {
    private int value;
    private String name;

    @Override
    public String toString() {
        return "MyPojo [value=" + value + ", name=" + name + "]";
    }
}
```

- In your course, almost every bean you created is a POJO.

2. What is a Java Bean?

- A **Java Bean** is a Java class that follows specific conventions, especially important in the past for frameworks like Enterprise Java Beans (EJB).
- **Three important constraints for a Java Bean:**
  1. **Public no-argument constructor:**
     The class must have a public constructor with no arguments (either explicit or default).
  2. **Getters and Setters:**
     The class must provide public methods to get and set the values of its properties.
  3. **Implements Serializable:**
     The class must implement the java.io.Serializable interface.
- **Example:**

```
public class MyJavaBean implements Serializable {
    private int value;
    private String name;

    public MyJavaBean() {} // No-arg constructor
```

```
    public int getValue() { return value; }
    public void setValue(int value) { this.value = value; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
}
```

- **Note:**
  - Java Beans were especially important for Java frameworks like EJB.
  - In modern Java development, the strict Java Bean conventions are not as important.

3. What is a Spring Bean?

- A **Spring Bean** is **any Java object that is managed by the Spring container** (a.k.a. Spring context or IOC container).
- **How do you get a Spring Bean?**
  - Define the object in your configuration or annotate it for component scanning.
  - The Spring container instantiates and manages the object's lifecycle.
- **Important:**
  - Any POJO can be a Spring Bean.
  - Java Beans can also be Spring Beans, but being a Java Bean is **not required** for being a Spring Bean.
- **Summary:**
  - If Spring manages the object, it is a Spring Bean.

4. Comparison Table

Feature	POJO	Java Bean	Spring Bean
Requirements	None	1. No-arg constructor 2. Getters & setters 3. Implements Serializable	None (except being managed by Spring)
Managed by Spring	No	No	Yes
Example Usage	Any Java class	Traditional frameworks, EJB	Spring Framework

5. Key Points from the Instructor

- **POJO:** Any Java class is a POJO by default—no constraints.
- **Java Bean:** Must have a public no-arg constructor, getters/setters, and implement Serializable.
- **Spring Bean:** Any Java object (POJO or Java Bean) that is **managed by Spring**.
- In modern Java/Spring development, you mostly deal with POJOs and Spring Beans.
- The term "Java Bean" is less important today, as EJB is rarely used.

In summary:

- **POJO:** Any simple Java object, no rules.
- **Java Bean:** A POJO with extra rules (no-arg constructor, getters/setters, Serializable).
- **Spring Bean:** Any Java object that Spring manages—could be a POJO, Java Bean,

or anything else.

You now understand the differences and relationships between these three important terms!

Question 3: How Can I List All Beans Managed by Spring Framework?

- **Goal:** List all beans (including your POJOs) that are currently managed by Spring.

How to do it:

- Use the Spring context's `getBeanDefinitionNames()` method:
  `String[] beanNames = context.getBeanDefinitionNames();`
- This returns an array of all bean names in the context (including internal Spring beans and your beans).
- To print them all, you can use Java functional programming (Java 8+):
  ```
  Arrays.stream(context.getBeanDefinitionNames())
        .forEach(System.out::println);
  ```
  - `Arrays.stream(...)` creates a stream of the bean names.
  - `.forEach(System.out::println)` prints each name.
- **Result:**
  When you run the application, you'll see output like:
  ```
  org.springframework.context.annotation.internalConfigurationAnnotationProcessor
  org.springframework.context.annotation.internalAutowiredAnnotationProcessor
  org.springframework.context.annotation.internalRequiredAnnotationProcessor
  org.springframework.context.annotation.internalCommonAnnotationProcessor
  helloWorldConfiguration
  name
  age
  person
  person2MethodCall
  person3Parameters
  address2
  address3
  ```
  - The first few are internal Spring beans.
  - The rest are your configuration and custom beans.
- **Other useful methods:**
  - `getBeanDefinitionCount()` – returns the total number of beans.
  - `getBeanDefinition(String name)` – gets definition for a specific bean.

Question 4: What If Multiple Matching Beans Are Available?

Scenario:

- You have more than one bean of the same type (e.g., two `Address` beans: `address2` and `address3`).
- If you try to retrieve a bean by type:

System.out.println(context.getBean(Address.class));

- **What happens?**
  - You get an exception:
    org.springframework.beans.factory.NoUniqueBeanDefinitionException:
    No qualifying bean of type 'com.in28minutes.learn_spring_framework.Address' available:
    expected single matching bean but found 2: address2,address3
  - Spring cannot decide which bean to return because there are multiple beans of type Address.
- The same happens for Person.class if you have multiple beans of that type:
  System.out.println(context.getBean(Person.class));
  - Exception will say it found person, person2MethodCall, and person3Parameters.

Summary of the Problem:

- **If there is more than one bean of a given type, and you ask Spring for a bean by type, Spring will throw an exception.**
  - This helps avoid ambiguity in dependency injection.

What's Next?

- **How can you give Spring a hint about which bean to use (prioritize one)?**
  - This will be covered in the next step (using @Primary, @Qualifier, etc.).

In summary:

- You can list all beans managed by Spring using context.getBeanDefinitionNames() and print them with a stream.
- If you have multiple beans of the same type, asking for one by type causes an exception—Spring doesn't know which to pick.
- You will learn how to resolve this ambiguity in the next lesson.

How Spring Handles Multiple Matching Beans – Solutions

Problem Recap

- When you have more than one bean of the same type (e.g., Address), and you try to inject or retrieve by type, Spring throws a NoUniqueBeanDefinitionException.
- Example: If both address2 and address3 are beans of type Address, asking for Address.class or injecting an Address parameter causes ambiguity.

How to Solve This?

1. Using @Primary

- **@Primary** tells Spring which bean to prefer when multiple beans of the same type exist.
- **How to use:**
  Annotate one of your beans with @Primary.
  @Bean(name = "address2")

```
@Primary
public Address address() {
    return new Address("Baker Street", "London");
}
```

- Now, whenever Spring needs an Address and can't decide (no further hints), it picks the @Primary bean.

2. Using @Qualifier

- **@Qualifier** gives even more control, letting you specify exactly which bean to inject, even if there's a primary.
- **How to use:**
  - Annotate the bean with a qualifier:

```
@Bean(name = "address3")
@Qualifier("address3qulifier")
public Address address3() {
    return new Address("Motinagar", "Hyderabad");
}
```

  - Use the same qualifier when injecting:

```
@Bean
public Person person5Qualifier(String name, int age,
@Qualifier("address3qulifier") Address address) {
    return new Person(name, age, address);
}
```

- This ensures that person5Qualifier gets the specific address3 bean, regardless of which bean is primary.

What Happens When You Run the Code?

- **Listing beans:** You can still list all beans as before.
- **Injecting by type:**
  - If you ask for context.getBean(Address.class), Spring injects the @Primary bean (address2).
  - If you ask for context.getBean(Person.class), Spring injects the @Primary bean (person4Parameters).
- **Injecting by qualifier:**
  - If you ask for person5Qualifier, Spring injects the bean with the matching qualifier (address3qulifier), which is address3.

Example Output (Simplified):

```
Nilesh
15
Person[name=Ravi, age=20, address=Address[firstLine=Miain Street, city=Utrecht]]
Person[name=Nilesh, age=15, address=Address[firstLine=Baker Street, city=London]]
Person[name=Nilesh, age=15, address=Address[firstLine=Motinagar, city=Hyderabad]]
Address[firstLine=Baker Street, city=London]
Person[name=Nilesh, age=15, address=Address[firstLine=Baker Street, city=London]]
Address[firstLine=Baker Street, city=London]
Person[name=Nilesh, age=15, address=Address[firstLine=Motinagar, city=Hyderabad]]
```

- Notice how beans are correctly injected and resolved, even with multiple candidates.

Summary of Solutions

- **@Primary**: Use when you want one bean to be the default for its type.
- **@Qualifier**: Use when you want to specify exactly which bean to inject, even if there's a primary.

Key Takeaways

- If you have multiple beans of the same type, you must tell Spring which one to use—otherwise, you'll get an error.
- @Primary and @Qualifier are the two main ways to resolve this ambiguity.
- These techniques work both for retrieving beans from the context and for auto-wiring dependencies in bean definitions.

In summary:
You now know how to resolve multiple matching bean issues in Spring using @Primary and @Qualifier. This lets you control exactly which beans are injected and retrieved, making your Spring applications robust and predictable.

Code:

App02HelloWorldSpring.java

```
package com.in28minutes.learn_spring_framework;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class App02HelloWorldSpring {

	public static void main(String[] args) {
		// 1: Launch a Spring Context

		var context = new
AnnotationConfigApplicationContext(HelloWorldConfiguration.class);

		// 2: Configure the things that we want spring to manage -
		// HelloWorldConfiguration - @Configuration
		// name - @Bean

		// 3: Retrieving Beans managed by Spring
		System.out.println(context.getBean("name"));

		System.out.println(context.getBean("age"));

		System.out.println(context.getBean("person"));

		System.out.println(context.getBean("person2MethodCall"));

		System.out.println(context.getBean("person3Parameters"));

		System.out.println(context.getBean("address2"));

		System.out.println(context.getBean(Person.class));

		System.out.println(context.getBean(Address.class));
```

```
			System.out.println(context.getBean("person5Qualifier"));


		}
}
```

HelloWorldConfiguration.java

```
package com.in28minutes.learn_spring_framework;

import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Primary;


// Eliminate verbosity in creating Java Beans
// Public accessor methods, constructor,
// equals , hashcode and toString are automatically created.
// Released in JDK 16.

record Person (String name, int age, Address address) {};


// Address - firstLine & city
record Address(String firstLine, String city) {};

@Configuration
public class HelloWorldConfiguration {

	@Bean
	public String name() {
		return "Nilesh";
	}

	@Bean
	public int age() {
		return 15;
	}

	@Bean
	public Person person() {
		return new Person("Ravi", 20, new Address("Miain Street", "Utrecht"));
	}

	@Bean
	public Person person2MethodCall() {
		return new Person(name(), age(), address());
	}

	@Bean
	public Person person3Parameters(String name, int age, Address address3) { //
name, age, address2
		return new Person(name, age, address3);
	}

//	No qualifying bean of type 'com.in28minutes.learn_spring_framework.Address'
//	available: expected single matching bean but found 2: address2,address3
	@Bean
	@Primary
	public Person person4Parameters(String name, int age, Address address) { //
```

```
name, age, address2
		return new Person(name, age, address);
	}

	@Bean
	public Person person5Qualifier(String name, int age,
@Qualifier("address3qulifier") Address address) { // name, age, address2
		return new Person(name, age, address);
	}

	@Bean(name = "address2")
	@Primary
	public Address address() {
		return new Address("Baker Street", "London");
	}

	@Bean(name = "address3")
	@Qualifier("address3qulifier")
	public Address address3() {
		return new Address("Motinagar", "Hyderabad");
	}
}

Output:
Nilesh
15
Person[name=Ravi, age=20, address=Address[firstLine=Miain Street, city=Utrecht]]
Person[name=Nilesh, age=15, address=Address[firstLine=Baker Street, city=London]]
Person[name=Nilesh, age=15, address=Address[firstLine=Baker Street, city=London]]
Address[firstLine=Baker Street, city=London]
Person[name=Nilesh, age=15, address=Address[firstLine=Baker Street, city=London]]
Address[firstLine=Baker Street, city=London]
Person[name=Nilesh, age=15, address=Address[firstLine=Motinagar, city=Hyderabad]]
```

Getting Started with Java Spring Framework - 6 - Using Spring Framework & Review

27 June 2025 23:55

Refactoring: Preparing Your Spring Project for the Gaming Application

1. Organize Code into Packages

- **Goal:** Clean up your project structure by organizing files into logical packages.
- **How to do it:**
  1. **Create a new package** for Hello World code:
     - Right-click your Java source folder (e.g., src/main/java).
     - Choose **New > Package**.
     - Name it com.in28minutes.learn_spring_framework.helloworld.
     - Click **Finish**.
  2. **Move your Hello World classes** (App02HelloWorldSpring.java and HelloWorldConfiguration.java) into this new package.
     - Drag and drop or use your IDE's "Move" refactor feature.
     - Confirm any dialogs about moving files.
- **Result:**
  - You now have a clean structure with a dedicated package for Hello World samples.
  - Your directory tree should look something like:

```
com.in28minutes.learn_spring_framework.helloworld
  |-- App02HelloWorldSpring.java
  |-- HelloWorldConfiguration.java
com.in28minutes.learn_spring_framework.game
  |-- (game-related files)
com.in28minutes.learn_spring_framework
  |-- (other files, if any)
```

2. Verify the Refactor

- Open your moved files and **organize imports** if your IDE prompts you.
- **Run your Hello World Spring app** (App02HelloWorldSpring) to ensure it works after the move.
- You should see the same output as before—**everything should work**.

3. Fix the Resource Leak Warning

- **Problem:**
  Your IDE warns:
  Resource leak: 'context' is never closed

This is because AnnotationConfigApplicationContext implements Closeable and should be closed to free resources.

- **Solution:**
  Use **Java's try-with-resources** statement, which will automatically close the context when done (even if an exception occurs).
- **How to refactor:**

```
try (var context = new
AnnotationConfigApplicationContext(HelloWorldConfiguration.class)) {
    // Your code to use the context goes here...
    System.out.println(context.getBean("name"));
    // ...etc
}
```

  - Remove the old variable declaration and put everything inside the try block.
  - This ensures context.close() is called automatically.
- **Result:**
  - The warning disappears.
  - Your code is cleaner and safer.

4. Prepare for the Next Step

- Now your Hello World Spring app is well-organized and resource-safe.
- **You are ready to apply the same ideas to your gaming application:**
  - Move/refactor your basic gaming app into its own package if you haven't already.
  - Prepare to launch the gaming app using Spring, just as you did with Hello World.

5. Exercise (Instructor's Suggestion)

- Before moving on, **try this yourself:**
  - Refactor the code into packages.
  - Apply try-with-resources for the Spring context.
  - Make sure everything runs fine.
- **Pause and experiment before continuing**—this practice will help you solidify your understanding.

Summary

- You refactored your project into packages for clarity.
- You resolved resource warnings by adopting try-with-resources.
- Your Hello World Spring app is now clean, organized, and ready for more advanced steps—just like your upcoming gaming application will be.

In summary:
You've improved your project's structure and code safety using packages and try-with-resources, setting the stage for integrating Spring into your gaming app. Take a moment to try these steps on your own before moving forward!

Springifying the Gaming App

1. Goal

- Transform the basic Java gaming app to use the **Spring Framework** for managing dependencies and object creation.

2. Copying and Renaming the Main Class

- **Copied** the main class from the previous basic Java version.
- **Renamed** it to App03GamingSpringBeans (since App02 already exists).
  - This class will become the Spring-based version of the gaming app.

3. Creating the Spring Configuration File

- **Created a new class**: GamingConfiguration.
- **Annotated** the class with @Configuration to let Spring know it's a configuration class.
  ```
  @Configuration
  public class GamingConfiguration { ... }
  ```
- **Purpose**: This file will define all the beans (managed objects) for the Spring container.

4. Defining Beans in the Configuration File

- **Started simple**: Focused only on PacmanGame first, left out other games for now.
  - This keeps the setup straightforward and avoids compilation errors.
- **Defined the first bean**:
  ```
  @Bean
  public GamingConsole game() {
      var game = new PacmanGame();
      return game;
  }
  ```
  - **What it does**: Registers a PacmanGame object as a Spring bean of type GamingConsole.

5. Setting Up the Spring Context in the Main Class

- **Created a Spring context** using AnnotationConfigApplicationContext with GamingConfiguration.class.
  ```
  try(var context = new AnnotationConfigApplicationContext(GamingConfiguration.class)) { ... }
  ```
- **Why try-with-resources?**
  - Avoids resource leak warnings.
  - Ensures the context is closed automatically, even if an exception occurs.

6. Retrieving and Using Beans

- **Got the GamingConsole bean** from the context and called the up() method.
  context.getBean(GamingConsole.**class**).up();
- **Ran the game using the GameRunner bean:**
  context.getBean(GameRunner.**class**).run();
- **No manual object creation** in the main class; all objects come from the Spring context.

7. Adding the GameRunner Bean

- **Defined another bean** in GamingConfiguration for GameRunner:

```
@Bean
public GameRunner gameRunner(GamingConsole game) {
    var gameRunner = new GameRunner(game);
    return gameRunner;
}
```

  - **Dependency Injection:**
    - The gameRunner bean takes a GamingConsole bean as a parameter.
    - Spring automatically injects the game() bean here.
    - This is constructor injection via method parameters in the configuration file.

8. Wiring Everything Together

- **Spring now manages both the game and gameRunner beans.**
- **The main app only retrieves and uses these beans—no new object creation or manual wiring.**
- **All configuration and dependencies are centralized in GamingConfiguration.**

9. Expected Output

- When running the application:
  - "up" (from the up() method on the game)
  - "Running game: PacmanGame" (or similar, depending on your GameRunner implementation)
  - Other game actions (e.g., up, down, left, right)

10. Summary of Spring Concepts Demonstrated

- **@Configuration**: Marks the class as a source of bean definitions.
- **@Bean**: Marks methods that return objects to be managed as beans by Spring.
- **Spring Context**: Created with AnnotationConfigApplicationContext, manages beans and their lifecycle.
- **Dependency Injection**: Beans can have dependencies (like GameRunner needing a GamingConsole), and Spring will inject them automatically.
- **try-with-resources**: Ensures the Spring context is closed automatically to avoid resource leaks.

11. Key Takeaways

- You learned how to transform a basic Java app to use Spring's powerful dependency injection and bean management.
- **Beans are defined in one place, dependencies are automatically resolved, and the main application is simplified.**
- This sets up a scalable, maintainable foundation for more complex applications using Spring.

In summary:
You started with a basic Java gaming app, created a Spring configuration class, defined beans, and used Spring's context and dependency injection to fully manage your objects. The main application now simply retrieves and uses beans from the context, with all lifecycle and wiring handled by Spring.

GamingConfiguration.java

```
package com.in28minutes.learn_spring_framework;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import com.in28minutes.learn_spring_framework.game.GameRunner;
import com.in28minutes.learn_spring_framework.game.GamingConsole;
import com.in28minutes.learn_spring_framework.game.PacmanGame;

@Configuration
public class GamingConfiguration {

	@Bean
	public GamingConsole game() {
		var game = new PacmanGame();
		return game;
	}

	@Bean
	public GameRunner gameRunner(GamingConsole game) {
		var gameRunner = new GameRunner(game);
		return gameRunner;
	}

}
```

App03GamingSpringBeans.java

```
package com.in28minutes.learn_spring_framework;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import com.in28minutes.learn_spring_framework.game.GameRunner;
import com.in28minutes.learn_spring_framework.game.GamingConsole;

public class App03GamingSpringBeans {

	public static void main(String[] args) {

		try(var context =
				new AnnotationConfigApplicationContext
				   (GamingConfiguration.class)) {
```

```
                context.getBean(GamingConsole.class).up();

                context.getBean(GameRunner.class).run();

            }
        }
}
```

Spring Gaming Application: Code Review and Concepts

1. Background and Progress

- **Earlier:**
  - All objects (game, game runner) were created manually in Java code and managed directly by the JVM.
  - You had to handle object creation and dependencies yourself.
- **Now:**
  - You've moved to using the **Spring Framework** to manage your objects (beans).
  - Spring handles object creation, wiring, and lifecycle management for you.

2. Configuration Class: Defining Beans

- **GamingConfiguration class:**
  - Marked with @Configuration to tell Spring this class contains bean definitions.

```
@Configuration
public class GamingConfiguration {
    @Bean
    public GamingConsole game() {
        var game = new PacmanGame();
        return game;
    }

    @Bean
    public GameRunner gameRunner(GamingConsole game) {
        var gameRunner = new GameRunner(game);
        return gameRunner;
    }
}
```

- **What's happening here?**
  - **PacmanGame bean:**
    - The game() method returns a PacmanGame instance, registered as a bean of type GamingConsole.
  - **GameRunner bean:**
    - The gameRunner() method takes a GamingConsole as a parameter.
    - Spring automatically injects the game bean here.
    - Returns a new GameRunner instance with the injected game.
  - **All beans are managed by Spring.**

3. Launching the Spring Context

- **App03GamingSpringBeans main class:**

```
public class App03GamingSpringBeans {
    public static void main(String[] args) {
        try (var context = new
AnnotationConfigApplicationContext(GamingConfiguration.class)) {
            context.getBean(GamingConsole.class).up();
            context.getBean(GameRunner.class).run();
        }
    }
}
```

- **What's happening here?**
  - **Creates the Spring context** using the configuration class.
  - **Uses try-with-resources** so the context is closed automatically (no resource leaks).
  - **Retrieves beans from the context:**
    - Gets the GamingConsole bean and calls its up() method.
    - Gets the GameRunner bean and calls its run() method.
  - **No manual instantiation** of game or game runner—Spring provides them.

4. Key Spring Concepts Demonstrated

- **Spring Container/Context:**
  - Manages the lifecycle and dependencies of beans.
- **@Configuration & @Bean:**
  - Used to declare and configure beans in a centralized place.
- **Dependency Injection:**
  - Spring automatically wires dependencies (e.g., injects the game bean into gameRunner()).
- **Decoupling:**
  - Main class is not tightly coupled to specific implementations.
- **Resource Management:**
  - try-with-resources ensures context is closed properly.

5. What's Next?

- **You have successfully moved from manual object management to Spring-managed beans.**
- **In the next steps:**
  - You'll learn how to simplify and enhance this code even further using more Spring features.

Summary Table

Step	What You Did
Manual Java objects	Created and managed all objects yourself
Spring Configuration	Defined beans in GamingConfiguration with @Bean methods
Spring Context Launch	Used AnnotationConfigApplicationContext to create context
Bean Retrieval	Got beans from context and invoked methods
Dependency Injection	Spring injected beans where needed

Resource Management	Used try-with-resources for safe context closure

In summary:
You started with manual Java object management and progressed to using Spring's context and beans. Now, Spring creates and wires your objects, and your main application simply uses them. This makes your code cleaner, more maintainable, and ready for future enhancements with Spring.

Spring Gaming App: Code Review and Deeper Questions

1. Where We Are Now

- **Current Setup:**
  - You have a GamingConfiguration class annotated with @Configuration defining beans for your game (PacmanGame) and the GameRunner.
  - In your main class (App03GamingSpringBeans), you create a Spring context, retrieve beans, and call their methods.
  - **Spring is now managing the beans** (objects) for you.

2. Key Observations and Questions Raised

Question Five:

Spring is managing objects and performing autowiring.
But aren't we writing the code to create objects?
How do we get Spring to create objects for us?

- **What's happening now?**
  - In GamingConfiguration, you are still **writing the code to create objects**:

```
public GamingConsole game() {
    var game = new PacmanGame(); // You create the object
    return game;
}
public GameRunner gameRunner(GamingConsole game) {
    var gameRunner = new GameRunner(game); // You create the object
    return gameRunner;
}
```

  - **Spring is managing the beans** (wiring them, controlling lifecycle), but the object instantiation (new PacmanGame(), new GameRunner(game)) is still your code.
- **The question:**
  - Is there a way to have Spring also create these objects for you, not just manage and wire them?

Question Six:

Is Spring framework really making things easy
or is it making it difficult?

- **Observation:**
  - If you compare the original, basic Java version (App01GamingBasicJava), the code

was simpler:
        - You just created objects and called their methods.
    - With Spring:
        - You have a configuration file with annotations and bean methods.
        - You have to set up and launch a Spring context.
- **The question:**
    - Does using Spring actually **simplify development**, or does it add complexity?
    - Is the extra configuration and setup worth it, compared to plain Java?

3. What Happens in Your Code (Summary)

- **You define how to create beans** in the configuration file.
- **Spring manages the beans** (dependency injection, lifecycle), but **you still write the constructors**.
- **Your main method** is decoupled from implementation details—it just asks the context for beans and uses them.

4. What's Next?

- **Upcoming lessons** will answer:
    - How can Spring create objects for you automatically, even without you writing `new`?
    - How does Spring's "component scanning" and "autowiring" make things easier?
    - Is Spring's complexity justified by the benefits it brings?

5. Key Takeaways from This Step

- You've transitioned from manual Java object creation to Spring-managed beans.
- Currently, you still write the code to instantiate objects in configuration.
- Spring handles wiring and lifecycle, but not yet full object creation.
- You're set up to learn about further Spring features that automate even more and simplify configuration.

In summary:
You're now using Spring to manage and wire your objects, but you still write code to instantiate them in configuration. The next steps will show how Spring can automate object creation even further, and will address whether Spring is truly simplifying your development process or adding unnecessary complexity.

Spring Fundamentals: Section Recap and What's Next

1. What You Learned in This Section

- **Tight vs. Loose Coupling**
    - Understood what tight coupling means and why it makes code less flexible.

    - Learned how to design code to be *loosely coupled* using interfaces.
- **Using Java Interfaces**
    - Saw how interfaces can help decouple components, making your code more modular and easier to maintain or extend.
- **Introducing the Spring Container**
    - Learned how to bring in the Spring container (a.k.a. Spring context).
    - Discovered how Spring manages objects (beans) for you.
- **Application Context**
    - Created and launched an application context using Spring.
- **Spring Basic Annotations**
    - Learned about and used important Spring annotations:
        - `@Configuration`: Marks a class as a source of bean definitions.
        - `@Bean`: Declares a method as a bean producer.
    - Used these annotations to define and manage beans in your application.
- **Other Spring Annotations**
    - Were introduced to a range of other Spring annotations that enhance configuration and wiring.
- **Autowiring**
    - One of the most important concepts covered.
    - Understood how Spring can automatically inject dependencies into your beans.
- **Java Bean vs. Spring Bean**
    - Clarified the difference between a Java Bean (a POJO following certain conventions) and a Spring Bean (any object managed by the Spring container).

2. Why These Concepts Matter

- **Foundation for Great Development**
    - Mastering these concepts is essential for building robust, maintainable Spring applications.
    - Helps not only in designing new applications but also in debugging and maintaining existing ones.
- **Hands-On, Visual Learning**
    - The approach is designed to be highly practical, visual, and repeatable for maximum clarity.
    - The instructor has iterated through examples multiple times for smooth learning.

3. What's Coming Next

- **Simplifying Bean Creation**
    - The next section will focus on making bean creation *much* simpler than the current approach.
- **More Spring Annotations**
    - You'll dive deeper into:
        - `@Primary`
        - `@Qualifier`
        - `@Component` and its variants
    - These annotations provide even more powerful and flexible ways to manage and inject dependencies.
- **Spring Terminology**

    - Will learn more about the terminology used in the Spring ecosystem, building a strong conceptual foundation.
- **Dependency Injection (DI)**
    - Will gain a deeper understanding of DI, including the different types supported by Spring.
- **Real-World Example**
    - The section will end with a practical, real-world Spring application example to see all concepts in action.

4. Encouragement and Next Steps

- **You're on the Right Track!**
    - Mastering these Spring fundamentals puts you well on your way to becoming a great developer.
    - The next steps will make your code even cleaner and more efficient.
- **Ready to Learn More?**
    - The upcoming section promises more hands-on examples and advanced concepts.
    - Get excited to see how Spring can make your development experience even better!

In summary:
You've built a solid foundation in Spring: understanding coupling, interfaces, the Spring container, basic annotations, autowiring, and the distinction between Java beans and Spring beans. Next, you'll simplify your code further, learn advanced annotations and DI techniques, and apply your knowledge in a real-world Spring project.

Congratulations on Your Progress!

1. Celebrating Your Achievement

- You've reached the **first milestone** of this course—well done!
- Not everyone even starts an online course, so your commitment is something to be proud of.
- **Keep up the great work!**

2. The Important Question: How to Remember What You Learn?

- Learning something new is valuable, but **retaining** that knowledge over time is the real challenge.
- Ask yourself:
    - How can I remember what I've learned one week, one month, or even one year from now?

3. Two Powerful Strategies for Long-Term Retention

A. Active Learning

- **Definition:**
  - Engage with the material actively, not passively.
  - One of the best ways is to **take notes** as you learn.
- **How to do it:**
  - Whenever you see or hear something interesting, **write it down in your own words.**
  - Summarize concepts, jot down questions, and highlight key points.
- **Why it works:**
  - Writing reinforces learning by making your brain process and organize information.

B. Regular Review

- **Definition:**
  - Don't just take notes—**review them regularly.**
- **How to do it:**
  - Set aside time every few days or each week to go back over your notes.
  - Refresh your memory on key concepts and revisit challenging topics.
- **Why it works:**
  - Your brain is wired to remember things you revisit frequently.
  - Regular review strengthens long-term retention and recall.

4. Personal Experience from the Instructor

- The instructor attributes the ability to:
  - Learn multiple programming languages, frameworks, and cloud platforms,
  - Maintain proficiency across technologies,
- **To the consistent habits of note-taking and regular review.**
- Old notes are still used as references today, showing the lasting value of this approach.

5. Encouragement

- You're on a great learning path!
- Continue using active learning and regular review to make your progress stick.
- The instructor is excited to see you succeed and will see you again soon in the course.

In summary:
To remember what you learn for the long term:

- **Take notes** as you learn (active learning).
- **Review your notes** regularly. These simple habits will help you retain knowledge, build expertise, and accelerate your learning journey.

Using Spring Framework to Create and Manage Your Java Objects

15 July 2025 02:09

Spring Bean Creation: Code Review and Key Insights

1. Background: Manual vs. Automatic Bean Creation

- **Current Situation:**
  - You have a class (App03GamingSpringBeans) that both launches the Spring context and defines beans using @Bean methods.
  - Example:

```
@Bean
public GamingConsole game() {
    var game = new PacmanGame();
    return game;
}

@Bean
public GameRunner gameRunner(GamingConsole game) {
    var gameRunner = new GameRunner(game);
    return gameRunner;
}
```

  - The main method launches the context and retrieves beans:

```
try(var context = new
AnnotationConfigApplicationContext(App03GamingSpringBeans.class
)) {
    context.getBean(GamingConsole.class).up();
    context.getBean(GameRunner.class).run();
}
```

  - **Spring manages and wires the beans, but you are still writing code to create them (new PacmanGame(), etc).**

2. Refactoring and Simplification Steps from the Lecture

- **Copied and Renamed the Project:**
  - Instructor suggests copying and renaming the project to keep a reference to the old version for comparison.

    - This is good practice when making major changes—so you always have a backup.
- **Cleanup:**
    - Deleted unnecessary files and packages (like HelloWorld and old gaming app files).
    - Kept only relevant files in the main package and the `game` package.
- **Combining Configuration and Launcher:**
    - Moved the `@Bean` methods from a separate configuration file into the main launcher class.
    - Removed the `public` modifier from the inner configuration class (since only one public class per file).
    - Deleted the old separate configuration class.
    - Now, **one class contains both the configuration and the main method**.
    - Updated the context initialization:
        - Uses `App03GamingSpringBeans.class` as the configuration source.
- **Testing:**
    - Ran the application to confirm everything still works after the refactor.

3. Key Observations

- **You Still Write Bean Creation Code:**
    - Even though Spring manages and injects the beans, you still manually instantiate them with `new` in your `@Bean` methods.
- **Single-Class Simplicity:**
    - The entire Spring configuration and application bootstrap are now in a single file.
    - This makes the project easier to understand for small/simple apps.
- **Spring Context Initialization:**
    - The class annotated with `@Configuration` and containing `@Bean` methods is passed to `AnnotationConfigApplicationContext` to bootstrap the context.

4. What's Next?

- **Automating Bean Creation Even Further:**
    - The instructor hints that the next steps will show how to let Spring **automatically create beans** for you—without needing explicit `@Bean` methods and `new` calls.
    - This will use advanced Spring features like **component scanning** and **stereotype annotations** (`@Component`, etc).

5. Takeaways and Transition

- You've learned:
    - How to define beans using `@Bean` methods (manual object creation, managed by Spring).
    - How to consolidate configuration and launching logic into a single class.
    - How to keep your workspace clean and focused by removing unnecessary files.
- **Next Steps:**

- Explore Spring's ability to discover and instantiate beans automatically, reducing boilerplate code and making configuration even simpler.

In summary:
You've refactored your project to put all Spring bean definitions and launch logic in a single class. While Spring manages the beans, you still write code to instantiate them. In the next section, you'll see how Spring can create beans automatically through component scanning, making your code even cleaner and more maintainable.

App03GamingSpringBeans.java

```
package com.in28minutes.learn_spring_framework;


import
org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import com.in28minutes.learn_spring_framework.game.GameRunner;
import com.in28minutes.learn_spring_framework.game.GamingConsole;
import com.in28minutes.learn_spring_framework.game.PacmanGame;

@Configuration
public class App03GamingSpringBeans {

	@Bean
	public GamingConsole game() {
		var game = new PacmanGame();
		return game;
	}

	@Bean
	public GameRunner gameRunner(GamingConsole game) {
		var gameRunner = new GameRunner(game);
		return gameRunner;
	}

	public static void main(String[] args) {

		try(var context =
				new AnnotationConfigApplicationContext
				   (App03GamingSpringBeans.class)) {

			context.getBean(GamingConsole.class).up();

			context.getBean(GameRunner.class).run();

		}
	}
}
```

Spring Bean Creation with

@Component and @ComponentScan

1. Goal: Let Spring Automatically Create Beans

- **Previously:**
  - Beans like PacmanGame were created manually in @Bean methods (using new).
  - You wanted Spring to not just manage, but *also create* beans automatically.

2. Using @Component for Automatic Bean Creation

- **What is @Component?**
  - Marks a class as a Spring-managed component.
  - Tells Spring: "You are responsible for creating an instance of this class."
- **How did you use it?**
  - Annotated PacmanGame with @Component:
    ```
    @Component
    public class PacmanGame implements GamingConsole { ... }
    ```
  - Now, Spring will automatically detect and instantiate PacmanGame as a bean.

3. How Does Spring Find Components?

- **Problem Encountered:**
  - Even after adding @Component, Spring **couldn't find** PacmanGame by default.
  - You got an error: NoSuchBeanDefinitionException for GamingConsole.
  - Why? Because Spring doesn't know where to look for components unless you tell it.

4. Solution: Use @ComponentScan

- **What is @ComponentScan?**
  - Tells Spring which packages to search for components (classes annotated with @Component).
- **How did you use it?**
  - Added @ComponentScan("com.in28minutes.learn_spring_framework.game") to your configuration:
    ```
    @Configuration
    @ComponentScan("com.in28minutes.learn_spring_framework.game")
    public class App03GamingSpringBeans { ... }
    ```
  - Now, Spring will scan the specified package, find PacmanGame, and create a bean for it.

5. Simplifying the Configuration

- You **removed** the manual @Bean method for GamingConsole/PacmanGame.
- You **kept** the @Bean for GameRunner, which takes a GamingConsole as a parameter.
- Spring now:
  - Automatically creates the PacmanGame bean using @Component.
  - Injects it into the GameRunner bean.

6. Verification & Output

- **Added a print statement** in the gameRunner bean method:
  System.out.println("Parameter: " + game);
- When running the application:
  - Confirms that the parameter received is an instance of PacmanGame created by Spring.
  - The application works: Spring creates and injects PacmanGame automatically.

7. Key Spring Concepts Demonstrated

- **@Component**: Declares a class as a candidate for Spring's auto-detection and bean creation.
- **@ComponentScan**: Instructs Spring on which packages to scan for components.
- **Automatic Dependency Injection**: Spring finds the @Component, creates the bean, and injects it wherever needed.
- **Reduced Boilerplate**: No need to write @Bean methods for every class—just use @Component and let Spring handle creation.

8. Summary Table

Step	What You Did
Add @Component	Marked PacmanGame for Spring's auto-detection and bean creation
Add @ComponentScan	Told Spring which package to scan for components
Remove manual @Bean	No need to write a @Bean for PacmanGame anymore
Keep GameRunner @Bean	Still used @Bean for wiring and to show parameter injection
Print parameter for check	Verified that Spring was injecting the right bean
Run and verify	Application worked, Spring handled everything automatically

9. Next Steps (as hinted by instructor)

- How to get Spring to create *even more* beans automatically (e.g., GameRunner).
- Further exploration of component scanning and auto-wiring.
- Making configuration even simpler.

In summary:
You've learned how to let Spring create beans automatically using @Component and @ComponentScan. You no longer need to manually instantiate every bean in configuration. Spring scans your packages, finds components, creates them, and wires them for you—making your code cleaner and easier to maintain.

PacmanGame.java

```
package com.in28minutes.learn_spring_framework.game;

import org.springframework.stereotype.Component;

@Component
public class PacmanGame implements GamingConsole{
	public void up() {
		System.out.println("up");
```

```
	}

	public void down() {
		System.out.println("down");
	}

	public void left() {
		System.out.println("left");
	}

	public void right() {
		System.out.println("right");
	}
}
```

App03GamingSpringBeans.java

```
package com.in28minutes.learn_spring_framework;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

import com.in28minutes.learn_spring_framework.game.GameRunner;
import com.in28minutes.learn_spring_framework.game.GamingConsole;

@Configuration
@ComponentScan("com.in28minutes.learn_spring_framework.game")
public class App03GamingSpringBeans {

	@Bean
	public GameRunner gameRunner(GamingConsole game) {
		System.out.println("Parameter: " + game);
		var gameRunner = new GameRunner(game);
		return gameRunner;
	}

	public static void main(String[] args) {

		try(var context =
				new AnnotationConfigApplicationContext
				   (App03GamingSpringBeans.class)) {

			context.getBean(GamingConsole.class).up();

			context.getBean(GameRunner.class).run();

		}
	}
}
```

Spring Bean Creation Fully Automated with @Component and @ComponentScan

1. Goal: Let Spring Automatically Create & Wire All Beans

- **Objective:**
  Move from manually writing @Bean methods to letting Spring automatically create and wire *all* your beans using annotations.
- **Result:**
  Much less code, greater simplicity, and true dependency injection!

2. Final Code Breakdown

a) Main Application Class

```
@Configuration
@ComponentScan("com.in28minutes.learn_spring_framework.game")
public class GamingAppLauncherApplication {

    public static void main(String[] args) {
        try(var context =
                new
AnnotationConfigApplicationContext(GamingAppLauncherApplication.class)) {
            context.getBean(GamingConsole.class).up();
            context.getBean(GameRunner.class).run();
        }
    }
}
```

- **What's happening?**
  - @Configuration: Marks this class as a Spring configuration source.
  - @ComponentScan("com.in28minutes.learn_spring_framework.game"): Tells Spring to scan the specified package for components to manage.
  - The main method launches the Spring context and retrieves beans for use.

b) PacmanGame Bean

```
@Component
public class PacmanGame implements GamingConsole {
    // Implements up(), down(), left(), right()
}
```

- **What's happening?**
  - @Component: Spring will automatically detect, instantiate, and manage this class as a bean.

c) GameRunner Bean

```
@Component
public class GameRunner {
    private GamingConsole game;

    public GameRunner(GamingConsole game) {
        this.game = game;
    }

    public void run() {
```

```
        // Runs the game
    }
}
```

- **What's happening?**
  - @Component: Marks this class as a Spring-managed bean.
  - **Constructor injection:** Spring detects that GameRunner needs a GamingConsole and provides the appropriate bean (PacmanGame).

3. Key Steps from the Lecture

- **Added @Component to both PacmanGame and GameRunner.**
  - Now, Spring automatically creates beans for both.
  - No need for manual @Bean methods or new statements.
- **Ensured both classes are in the package specified by @ComponentScan.**
  - If you ever get a "NoSuchBeanDefinitionException," check your package scan path and @Component annotations!
- **Renamed the main class for clarity (to GamingAppLauncherApplication).**
  - Makes the purpose of the class clearer.

4. What Did You Achieve?

- **Spring now creates and wires all objects for you.**
  - You don't have to write @Bean methods or manual constructors.
  - All you do is:
    - Annotate your classes with @Component.
    - Make sure they're in the scanned package.
    - Use constructor injection for dependencies.
- **Your code is much cleaner and easier to maintain.**
- **Spring performs component scanning, object creation, and autowiring automatically.**

5. Troubleshooting

- If something isn't working:
  - **Check that your @Component annotations are present.**
  - **Make sure the @ComponentScan path matches the package of your beans.**
  - Both PacmanGame and GameRunner should be in the scanned package and annotated with @Component.

6. Further Exploration

- The instructor suggests:
  - Try adding @Component to other game classes (like MarioGame or SuperContraGame).
  - See what happens if you have multiple beans of the same type—how does Spring choose which one to inject?
  - This leads to learning about @Primary, @Qualifier, and more advanced dependency injection options.

7. Key Spring Concepts Used

Concept	Purpose/Effect
@Component	Marks a class as a Spring-managed bean
@ComponentScan	Tells Spring where to look for @Component classes to create as beans
@Configuration	Declares a class as a Spring configuration source
Constructor Injection	Spring wires dependencies automatically via constructors

8. Summary

- Spring now:
  - **Scans** your code for components,
  - **Creates** bean instances,
  - **Wires** dependencies automatically,
  - **Manages** the entire application lifecycle.
- Your code is now minimal and elegant—just add @Component and constructor dependencies, and Spring does the rest!

In summary:
You have fully automated bean creation and dependency injection in your Spring app using @Component and @ComponentScan. Your main class simply launches the context and uses the beans; Spring handles the rest, making your codebase much simpler and more powerful.

PacmanGame.java

```
package com.in28minutes.learn_spring_framework.game;

import org.springframework.stereotype.Component;

@Component
public class PacmanGame implements GamingConsole{
	public void up() {
		System.out.println("up");
	}

	public void down() {
		System.out.println("down");
	}

	public void left() {
		System.out.println("left");
	}

	public void right() {
		System.out.println("right");
	}
}
```

GameRunner.java

```
package com.in28minutes.learn_spring_framework.game;

import org.springframework.stereotype.Component;

@Component
public class GameRunner {

	private GamingConsole game;
```

```
	public GameRunner(GamingConsole game) {

		this.game = game;
	}

	public void run() {
		System.out.println("Running game: " + game);
		game.up();
		game.down();
		game.left();
		game.right();
	}
}
```

GamingAppLauncherApplication.java

```
package com.in28minutes.learn_spring_framework;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

import com.in28minutes.learn_spring_framework.game.GameRunner;
import com.in28minutes.learn_spring_framework.game.GamingConsole;

@Configuration
@ComponentScan("com.in28minutes.learn_spring_framework.game")
public class GamingAppLauncherApplication {

	public static void main(String[] args) {

		try(var context =
				new AnnotationConfigApplicationContext
					(GamingAppLauncherApplication.class)) {

			context.getBean(GamingConsole.class).up();

			context.getBean(GameRunner.class).run();

		}
	}
}
```

Spring Component Scanning & Bean Creation: Code Review and Key Concepts

1. What Was Achieved?

- You refactored your code so that **Spring automatically creates and wires your beans** using @Component and @ComponentScan.
- Your main application class simply launches the Spring context and uses the beans—no manual bean creation code is required.

2. Key Code Changes: Step-by-Step Review

A. Added @Component to Classes

- @Component was added to both PacmanGame and GameRunner:
  ```
  @Component
  public class PacmanGame implements GamingConsole { ... }

  @Component
  public class GameRunner { ... }
  ```
- **Purpose:**
  - Marks these classes as candidates for Spring's automatic bean creation and management.
  - Spring will scan for classes with @Component and instantiate them as beans.

B. Added @ComponentScan to the Configuration

- In your main class, you annotated with both @Configuration and @ComponentScan:
  ```
  @Configuration
  @ComponentScan("com.in28minutes.learn_spring_framework.game")
  public class GamingAppLauncherApplication { ... }
  ```
- **Purpose:**
  - Tells Spring to scan the specified package (com.in28minutes.learn_spring_framework.game) for components.
  - Ensures Spring knows where to look for @Component-annotated classes.

C. How Spring Creates and Wires Beans

- When you launch GamingAppLauncherApplication, Spring:
  1. **Scans** the package specified in @ComponentScan.
  2. **Finds** all classes with @Component (here: PacmanGame and GameRunner).
  3. **Creates beans** for these classes.
  4. **Handles dependency injection** (e.g., injects GamingConsole into GameRunner via the constructor).

D. Running the Application

- The main method retrieves and uses the beans:
  ```
  context.getBean(GamingConsole.class).up();
  context.getBean(GameRunner.class).run();
  ```
- **By the time this code runs, the beans are ready and fully wired by Spring.**

3. Why These Changes Matter

- **No More Manual Bean Creation:**
  - You no longer need to write @Bean methods or instantiate objects with new yourself.
- **Automatic Discovery:**
  - Spring finds and manages your beans automatically based on simple annotations.
- **Cleaner, More Maintainable Code:**
  - Less boilerplate, fewer configuration files, and no manual wiring.
- **True Dependency Injection:**
  - Spring injects dependencies (like GamingConsole into GameRunner) automatically.

4. Troubleshooting Tip

- If the application doesn't work:
  - Check that both PacmanGame and GameRunner are in the package specified by @ComponentScan.
  - Ensure both classes are annotated with @Component.

5. Summary Table

Step	Description
@Component on beans	Marks classes for Spring to instantiate as beans
@ComponentScan on config	Tells Spring which package to scan for components
Beans auto-created by Spring	No manual instantiation or @Bean methods needed
Beans auto-wired by Spring	Dependencies are injected via constructors automatically
Main uses beans from context	Beans are ready to use via context.getBean()

6. In Summary

- **Spring is now doing all the heavy lifting:**
  - Scanning for beans, creating them, and wiring dependencies.
- **Your code is concise, focused, and leverages the full power of Spring's auto-configuration.**
- **You're ready for even more advanced Spring features!**

In summary:
You've used @Component and @ComponentScan so that Spring automatically creates and wires your beans. No more manual configuration is required—just add the annotations and let Spring do the rest!

PacmanGame.java

```
package com.in28minutes.learn_spring_framework.game;

import org.springframework.stereotype.Component;

@Component
public class PacmanGame implements GamingConsole{
	public void up() {
		System.out.println("up");
	}

	public void down() {
		System.out.println("down");
	}

	public void left() {
		System.out.println("left");
	}

	public void right() {
		System.out.println("right");
	}
}
```

GameRunner.java

```
package com.in28minutes.learn_spring_framework.game;

import org.springframework.stereotype.Component;

@Component
public class GameRunner {

	private GamingConsole game;

	public GameRunner(GamingConsole game) {

		this.game = game;
	}

	public void run() {
		System.out.println("Running game: " + game);
		game.up();
		game.down();
		game.left();
		game.right();
	}
}
```

GamingAppLauncherApplication.java

```
package com.in28minutes.learn_spring_framework;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

import com.in28minutes.learn_spring_framework.game.GameRunner;
import com.in28minutes.learn_spring_framework.game.GamingConsole;

@Configuration
@ComponentScan("com.in28minutes.learn_spring_framework.game")
public class GamingAppLauncherApplication {

	public static void main(String[] args) {

		try(var context =
				new AnnotationConfigApplicationContext
				   (GamingAppLauncherApplication.class)) {

			context.getBean(GamingConsole.class).up();

			context.getBean(GameRunner.class).run();

		}
	}
}
```

Spring Dependency Injection: Handling Multiple Bean Candidates with @Primary and @Qualifier

1. The Scenario: Multiple Beans of the Same Type

- You have several classes implementing the GamingConsole interface:
  - PacmanGame
  - MarioGame
  - SuperContraGame
- Each class is annotated with @Component, so Spring will auto-create beans for all of them (since they are also in the scanned package).
- **Problem:** When Spring tries to inject a GamingConsole into GameRunner, it finds multiple candidates and doesn't know which to choose.

2. The Error: NoUniqueBeanDefinitionException

- When running the app with multiple @Component beans for the same interface, Spring throws:
  - **NoUniqueBeanDefinitionException**
    "No qualifying bean of type 'GamingConsole' available: expected single matching bean but found 2 (MarioGame, PacmanGame)"
- **Reason:** Spring cannot decide which bean to inject when there are multiple candidates for the same type.

3. Solution 1: Using @Primary

- **What is @Primary?**
  - You annotate one bean (e.g., MarioGame) with @Primary:
    ```
    @Component
    @Primary
    public class MarioGame implements GamingConsole { ... }
    ```
  - **Effect:** If multiple beans are found, Spring will choose the @Primary bean for injection by default.
- **Result:**
  - When you run the app, MarioGame is injected into GameRunner and its methods are called.

4. Solution 2: Using @Qualifier

- **What if you want to inject a specific bean, not the primary?**
  - Use @Qualifier to specify which bean to inject, by giving a unique qualifier name.
- **How to use:**
  - Annotate the target bean:
    ```
    @Component
    @Qualifier("SuperContraGameQualifier")
    public class SuperContraGame implements GamingConsole { ... }
    ```
  - In the GameRunner constructor, use @Qualifier on the parameter:
    ```
    public GameRunner(@Qualifier("SuperContraGameQualifier")
    GamingConsole game) {
        this.game = game;
    ```

}

- **Effect:**
  - Spring injects the bean that matches the given qualifier, regardless of @Primary settings.
  - When you run the app, SuperContraGame is injected and its methods are called.

5. How It Works in Code

- All beans are created by Spring thanks to @ComponentScan.
- @Primary tells Spring which bean to use by default when multiple candidates exist.
- @Qualifier lets you override the default and choose exactly which bean to inject for a specific dependency.

6. Troubleshooting

- If Spring cannot find a unique bean to inject, check:
  - Are all candidate classes annotated with @Component?
  - Are they in the package scanned by @ComponentScan?
  - Have you set @Primary or @Qualifier appropriately?
- If you want to select a bean by name, always use @Qualifier with a matching string.

7. Summary Table

Annotation	Where to Use	Purpose	Example Use
@Component	On class	Marks class as Spring-managed bean	@Component
@Primary	On one bean/class	Default bean when multiple candidates exist	@Component @Primary on MarioGame
@Qualifier	On bean & injection	Pick a specific bean when multiple are available	@Qualifier("SuperContraGameQualifier")

8. Key Takeaways

- **Spring needs help** when there are multiple beans for the same interface.
- Use @Primary to set a default bean for autowiring.
- Use @Qualifier to inject a specific bean by name, overriding @Primary.
- **Constructor injection** with @Qualifier is the most explicit and robust way to control which bean gets injected.

In summary:
When you have multiple beans of the same interface, Spring cannot choose automatically and throws an error.

- Mark one as @Primary to set the default.
- Use @Qualifier to inject a specific bean when needed. This gives you full control over Spring's dependency injection, even in complex scenarios.

PacmanGame.java

```
package com.in28minutes.learn_spring_framework.game;

import org.springframework.stereotype.Component;
```

```
@Component
public class PacmanGame implements GamingConsole{
	public void up() {
		System.out.println("up");
	}

	public void down() {
		System.out.println("down");
	}

	public void left() {
		System.out.println("left");
	}

	public void right() {
		System.out.println("right");
	}
}
```

GameRunner.java

```
package com.in28minutes.learn_spring_framework.game;

import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Component;

@Component
public class GameRunner {

	private GamingConsole game;

	public GameRunner(@Qualifier("SuperContraGameQualifier") GamingConsole game) {

		this.game = game;
	}

	public void run() {
		System.out.println("Running game: " + game);
		game.up();
		game.down();
		game.left();
		game.right();
	}
}
```

MarioGame.java

```
package com.in28minutes.learn_spring_framework.game;

import org.springframework.context.annotation.Primary;
import org.springframework.stereotype.Component;

@Component
@Primary
public class MarioGame implements GamingConsole{
	public void up() {
		System.out.println("Jump");
	}

	public void down() {
		System.out.println("Go into a hole");
```

```
	}

	public void left() {
		System.out.println("Go back");
	}

	public void right() {
		System.out.println("Accelerate");
	}
}
```

SuperContraGame.java

```
package com.in28minutes.learn_spring_framework.game;

import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Component;

@Component
@Qualifier("SuperContraGameQualifier")
public class SuperContraGame implements GamingConsole{
	public void up() {
		System.out.println("up");
	}

	public void down() {
		System.out.println("Sit down");
	}

	public void left() {
		System.out.println("Go back");
	}

	public void right() {
		System.out.println("Shoot a bullet");
	}
}
```

GamingAppLauncherApplication.java

```
package com.in28minutes.learn_spring_framework;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

import com.in28minutes.learn_spring_framework.game.GameRunner;
import com.in28minutes.learn_spring_framework.game.GamingConsole;

@Configuration
@ComponentScan("com.in28minutes.learn_spring_framework.game")
public class GamingAppLauncherApplication {

	public static void main(String[] args) {

		try(var context =
				new AnnotationConfigApplicationContext
				   (GamingAppLauncherApplication.class)) {

			context.getBean(GamingConsole.class).up();

			context.getBean(GameRunner.class).run();
```

```
            }
        }
}
```

Spring Dependency Injection: @Primary vs @Qualifier – Which to Use, and When?

1. Scenario: Multiple Implementations and Autowiring

- **Context:**
  You have an interface, SortingAlgorithm, with several implementations:
  - QuickSort
  - BubbleSort
  - RadixSort
- **Usage:**
  - ComplexAlgorithm and AnotherComplexAlgorithm are components that need a SortingAlgorithm.
  - Both use dependency injection (@Autowired).

2. What Does @Primary Do?

- **Purpose:**
  - Marks a bean as the *preferred* candidate when multiple beans are available for autowiring.
- **How it works:**
  - If multiple beans of the same type exist and *one* is marked with @Primary, Spring will inject the primary bean unless told otherwise.
- **Example:**
  - If you have five sorting algorithms and only QuickSort is marked @Primary, then:
    ```
    @Component
    @Primary
    public class QuickSort implements SortingAlgorithm { ... }
    ```
    - Any injection point with just @Autowired (no @Qualifier) will get QuickSort.

3. What Does @Qualifier Do?

- **Purpose:**
  - Lets you specify *exactly* which bean to inject, by name or custom qualifier.
- **How it works:**
  - Annotate the bean and the injection point with a qualifier string.
    ```
    @Component
    @Qualifier("RadixSortQualifier")
    public class RadixSort implements SortingAlgorithm { ... }

    public class AnotherComplexAlgorithm {
       @Autowired
       @Qualifier("RadixSortQualifier")
    ```

```
    private SortingAlgorithm sortingAlgorithm;
}
```

- **Effect:**
  - Spring injects the bean with the matching qualifier, *regardless of @Primary*.

4. Which One Should You Use?

- **Decide from the perspective of the consuming class:**
  - If the class is happy to use the "best" or "default" implementation, rely on @Primary (or just @Autowired if only one bean exists).
  - If the class *must* use a specific implementation, use @Qualifier.
- **Examples:**
  - ComplexAlgorithm uses just @Autowired:
    - Spring injects the @Primary bean (e.g., QuickSort).
  - AnotherComplexAlgorithm uses @Qualifier("RadixSortQualifier"):
    - Spring injects RadixSort regardless of what is marked as @Primary.

5. Priority: Qualifier vs. Primary

- **@Qualifier always has higher priority than @Primary.**
  - If a qualifier is specified at the injection point, Spring uses the matching bean, even if another bean is marked as @Primary.

6. Bean Name as Qualifier

- If you don't specify a custom qualifier, you can use the bean name (class name with a lowercase first letter) as the qualifier.
  - For RadixSort, the default bean name is radixSort.
  - So you could write:

```
@Autowired
@Qualifier("radixSort")
private SortingAlgorithm sortingAlgorithm;
```

  - This injects the RadixSort bean even without a custom @Qualifier on the bean.

7. Summary Table

Use Case	Annotation(s)	Effect
Preferred default bean	@Primary	Used by default for injection when multiple beans exist
Specific bean required	@Qualifier	Explicitly injects the named bean
Qualifier + Primary	Both present	@Qualifier takes precedence over @Primary
By bean name	@Qualifier("name")	Uses default bean name if no custom qualifier specified

8. Key Takeaways

- **Use @Primary** for general defaulting—when you want to set the default implementation.
- **Use @Qualifier** when you need a specific bean for a specific injection point.
- **Always think from the perspective of the class using the dependency.**
- **Remember:**
  - @Qualifier > @Primary in priority.

- If no `@Qualifier`, `@Primary` is used (if present).
- If neither is present and multiple beans exist, you get a `NoUniqueBeanDefinitionException`.

In summary:

- Use `@Primary` for default preference, and `@Qualifier` for explicit selection.
- `@Qualifier` always overrides `@Primary`.
- You can use the bean name as a qualifier if you don't define a custom one.
- Choose based on the requirements of the class using the dependency.

Spring Context Launchers in Different Packages: Refactored Structure and Concepts

1. What Did You Do?

- You now have **two different launcher classes**, each in its own package:
  - `DepInjectionLauncherApplication` in `com.in28minutes.learn_spring_framework.examples.a1`
  - `SimpleSpringContextLauncherApplication` in `com.in28minutes.learn_spring_framework.examples.a0`
- Both classes are annotated with `@Configuration` and `@ComponentScan`.
- Both launch a Spring context and print all the bean names created by the context.

2. How Does It Work?

a. @Configuration

- Marks the class as a source of bean definitions for the Spring context.

b. @ComponentScan

- Tells Spring to scan the current package (and its subpackages) for classes annotated with `@Component`, `@Service`, `@Repository`, or `@Controller`.
- **Default behavior:** If you don't specify a base package, Spring scans the package where the launcher class resides.

c. Creating the Context

- Each launcher starts a new `AnnotationConfigApplicationContext` using itself as the configuration class.
- This initializes a Spring context, scanning for components and creating beans.

d. Printing Bean Definitions

- `context.getBeanDefinitionNames()` lists all the beans (including Spring internal ones and any found in your packages).
- `Arrays.stream(...).forEach(System.out::println);` prints each bean name to the console.

3. Why Two Launchers?

- **Separation of Concerns:**
  Each launcher is responsible for its own context and component scanning in its own package.
- **Modularity:**
  You can test or experiment with different configurations, components, or features in isolation.
- **Learning Purpose:**
  Helps you see which beans are created in each context, and how Spring's classpath scanning is affected by package structure.

4. Key Spring Concepts Demonstrated

Concept	Description
@Configuration	Marks the class as a configuration source for the Spring context
@ComponentScan	Scans the current package and subpackages for Spring-managed beans
Bean Definition	Every Spring-managed object is called a "bean" and has a unique name in the context
Context Isolation	Each launcher creates a separate context with its own scanned beans
Package Structure	The package where the launcher resides determines the default scope of component scanning

5. What Can You Experiment With Next?

- **Add @Component classes in each package** and see which beans show up for each launcher.
- **Specify a base package in @ComponentScan** to control which packages are scanned.
- **Observe how context isolation works** by creating beans with the same name/type in different packages.

6. Example Output

When you run either launcher, you'll see a list of bean names, including Spring's internal beans and any @Component classes found in the package (and subpackages) of the launcher.

7. Summary Table

Launcher Class	Package	Scanned Packages (by default)
SimpleSpringContextLauncherApplication	com.in28minutes.learn_spring_framework.examples.a0	com.in28minutes.learn_spring_framework.examples.a0 and subpackages
DepInjectionLauncherApplication	com.in28minutes.learn_spring_framework.examples.a1	com.in28minutes.learn_spring_framework.examples.a1 and subpackages

In summary:
You now have two independent Spring launcher applications in different packages. Each creates its own Spring context, scans its own package for components, and prints all bean names—demonstrating modularity, package-based scanning, and

context isolation in Spring.

DepInjectionLauncherApplication.java

```
package com.in28minutes.learn_spring_framework.examples.a1;

import java.util.Arrays;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan
public class DepInjectionLauncherApplication {

	public static void main(String[] args) {

		try(var context =
				new AnnotationConfigApplicationContext
				   (DepInjectionLauncherApplication.class)) {


	Arrays.stream(context.getBeanDefinitionNames()).forEach(System.out::println);
		}
	}
}
```

SimpleSpringContextLauncherApplication.java

```
package com.in28minutes.learn_spring_framework.examples.a0;

import java.util.Arrays;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan
public class SimpleSpringContextLauncherApplication {

	public static void main(String[] args) {

		try(var context =
				new AnnotationConfigApplicationContext
				   (SimpleSpringContextLauncherApplication.class)) {


	Arrays.stream(context.getBeanDefinitionNames()).forEach(System.out::println);
		}
	}
}
```

Spring Dependency Injection: Field Injection Example

1. Goal of the Step

- Explore **three types of Dependency Injection** in Spring:
  1. **Constructor-based** (not covered here)
  2. **Setter-based** (not covered here)
  3. **Field-based (this example)**

2. Example Setup

- **Main Classes:**
  - YourBusinessClass
    - Depends on Dependency1 and Dependency2
  - Dependency1
  - Dependency2
- **All classes are annotated with @Component**, so Spring will automatically detect and create beans for them.

3. How the Example Works

A. Bean Creation

- Each class (YourBusinessClass, Dependency1, Dependency2) has @Component.
- Because of @ComponentScan (with no base package specified), **Spring scans the current package** and finds all components to create as beans.

B. Field Injection

- In YourBusinessClass:

```
@Component
class YourBusinessClass {
    @Autowired
    Dependency1 dependency1;
    @Autowired
    Dependency2 dependency2;
    ...
}
```

  - The @Autowired annotation tells Spring to **inject the required dependencies directly into the fields** (field injection).
  - No setters or constructors are needed for this type of injection.
  - Spring uses *reflection* to set the fields after the object is created.

C. Verifying Injection

- The toString() method in YourBusinessClass prints out the injected dependencies:

```
public String toString() {
    return "Using " + dependency1 + " and " + dependency2;
}
```

- The main method retrieves the bean and prints it:

```
System.out.println(context.getBean(YourBusinessClass.class));
```

- **Expected output:**
  "Using com.in28minutes.learn_spring_framework.examples.a1.Dependency1@xxxx and com.in28minutes.learn_spring_framework.examples.a1.Dependency2@yyyy"

4. How Spring Knows What to Inject

- **Spring Context Launch:**
  - The context is launched with DepInjectionLauncherApplication, which has @Configuration and @ComponentScan.
  - All classes with @Component in the package are detected and instantiated as beans.
- **Field Injection:**
  - When Spring creates the YourBusinessClass bean, it automatically finds and injects instances of Dependency1 and Dependency2 into the fields marked with @Autowired.

5. Key Points from the Lecture

- **Without @Autowired**, the dependencies would remain null.
- **With @Autowired**, Spring injects the dependencies automatically.
- **Field Injection** is simple and concise, but:
  - It relies on reflection.
  - It is less explicit than constructor or setter injection.
- **Best Practice Note:**
  - For real-world projects, it's often recommended to use constructor injection for required dependencies, as it's more explicit and testable.

6. Spring Concepts Demonstrated

Concept	Where/How Used
@Component	On all classes to mark them as beans for Spring to manage
@Autowired	On fields in YourBusinessClass for field injection
@ComponentScan	On config class to auto-detect components in the package
Field Injection	Dependencies injected directly into fields
Spring Context Launch	Using AnnotationConfigApplicationContext

7. Summary Table

Step	What Happened
Add @Component	Marked each class for bean creation
Add @Autowired	Marked fields for dependency injection
Launch Context	Spring scanned package, created beans, did injection
Print Bean	Showed injected dependencies via toString()

8. What's Next?

- In the next steps, you will explore:
  - **Constructor-based injection**
  - **Setter-based injection**

- This will help you compare the different approaches and understand when to use each.

In summary:
You've learned how to use **field injection** in Spring by marking dependencies with @Autowired on fields in your component class. Spring automatically creates and injects the required beans, making dependency management easy and concise.

DepInjectionLauncherApplication.java

```
package com.in28minutes.learn_spring_framework.examples.a1;

import java.util.Arrays;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.stereotype.Component;

@Component
class YourBusinessClass {

	@Autowired
	Dependency1 dependency1;

	@Autowired
	Dependency2 dependency2;

	public String toString() {
		return "Using " + dependency1 + " and " + dependency2;
	}
}

@Component
class Dependency1 {

}

@Component
class Dependency2 {

}


@Configuration
@ComponentScan
public class DepInjectionLauncherApplication {

	public static void main(String[] args) {

		try(var context =
				new AnnotationConfigApplicationContext
				   (DepInjectionLauncherApplication.class)) {


	Arrays.stream(context.getBeanDefinitionNames()).forEach(System.out::println);

				System.out.println(context.getBean(YourBusinessClass.class));
```

```
        }
    }
}
```

Spring Dependency Injection: Field, Setter, and Constructor Injection

1. Overview: Types of Dependency Injection in Spring

Spring supports three primary ways to inject dependencies into your beans:

1. **Field Injection**
2. **Setter Injection**
3. **Constructor Injection**

2. Code Example Structure

- **YourBusinessClass**
  - Depends on Dependency1 and Dependency2
  - Demonstrates all three injection types (field, setter, constructor)
- **Dependency1** and **Dependency2**
  - Simple component beans (no dependencies)
- **Configuration Class**
  - Uses @ComponentScan to automatically find and instantiate components

3. Field Injection

- **How?**
  Place @Autowired directly on fields.
  ```
  @Autowired
  Dependency1 dependency1;
  @Autowired
  Dependency2 dependency2;
  ```
- **Effect:**
  Spring sets the fields via reflection after object creation.
- **Pros:**
  - Simple, concise.
- **Cons:**
  - Less explicit (harder to see dependencies).
  - Harder to test (fields may be private).

4. Setter Injection

- **How?**
  Place @Autowired on setter methods.
  ```
  @Autowired
  public void setDependency1(Dependency1 dependency1) {
  ```

```
    System.out.println("Setter Injection - setDependency1");
    this.dependency1 = dependency1;
  }
  @Autowired
  public void setDependency2(Dependency2 dependency2) {
    System.out.println("Setter Injection - setDependency2");
    this.dependency2 = dependency2;
  }
```

- **Effect:**
  Spring calls these methods after creating the bean, injecting the dependencies.
- **Verification:**
  Add print statements inside setters; observe output to confirm setter injection.
- **Pros:**
  - Can inject optional dependencies.
  - Easier for testing/mocking.
- **Cons:**
  - Bean can be in an inconsistent state before all setters are called.

5. Constructor Injection

- **How?**
  Use a constructor with parameters for all dependencies. You can (but don't have to) add @Autowired.

```
// @Autowired // Not required for a single constructor
public YourBusinessClass(Dependency1 dependency1, Dependency2
dependency2) {
  System.out.println("Constructor Injection - YourBusinessClass");
  this.dependency1 = dependency1;
  this.dependency2 = dependency2;
}
```

- **Effect:**
  Spring uses the constructor to create the bean and injects dependencies as arguments.
- **Verification:**
  Add a print statement inside the constructor; observe output to confirm constructor injection.
- **Pros:**
  - All dependencies are set at creation; bean is always fully initialized.
  - Easy to make dependencies final.
  - Encouraged by Spring (best practice).
- **Cons:**
  - Slightly more boilerplate for classes with many dependencies.

6. Key Insights from the Lecture

- **You can switch between injection types** simply by commenting/uncommenting the relevant code and running the app.
- **Spring's default behavior:**
  - If there is only one constructor, Spring will use it for injection even without @Autowired.
- **Order of injection (if multiple are present):**
  1. Constructor Injection

2. Setter Injection
3. Field Injection

7. Which Injection Should You Use?

- **Spring's Recommendation:**
  Constructor Injection is preferred.
- **Why?**
  - All dependencies are provided at creation.
  - The bean is fully initialized after construction.
  - Encourages immutability and easier testing.
  - No need for @Autowired if there's only one constructor.

8. Summary Table

Type	How to Use	When to Use / Pros	Cons
Field	@Autowired on field	Quick, concise for simple cases	Less explicit, harder to test
Setter	@Autowired on setter	For optional dependencies, testable	Bean not always consistent
Constructor	Constructor with params	**Preferred!** Immutable, explicit	More boilerplate

9. What Did You See in Output?

- With **Setter Injection**: Print statements from setters when beans are created.
- With **Constructor Injection**: Print statement from constructor when the bean is created.
- In both cases, the toString() output confirms that dependencies are injected.

In summary:
You explored and compared field, setter, and constructor injection in Spring. **Constructor injection** is preferred by the Spring team for being explicit, robust, and ensuring beans are always fully initialized. Spring will use the constructor for injection even if @Autowired is omitted (if only one constructor exists).

DepInjectionLauncherApplication.java

```
package com.in28minutes.learn_spring_framework.examples.a1;

import java.util.Arrays;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.stereotype.Component;

@Component
class YourBusinessClass {

	Dependency1 dependency1;

	Dependency2 dependency2;
```

```
//	@Autowired
	public YourBusinessClass(Dependency1 dependency1, Dependency2 dependency2) {
		super();
		System.out.println("Constructor Injection - YourBusinessClass ");
		this.dependency1 = dependency1;
		this.dependency2 = dependency2;
	}

//	@Autowired
//	public void setDependency1(Dependency1 dependency1) {
//		System.out.println("Setter Injection - setDependency1 ");
//		this.dependency1 = dependency1;
//	}
//
//	@Autowired
//	public void setDependency2(Dependency2 dependency2) {
//		System.out.println("Setter Injection - setDependency2 ");
//		this.dependency2 = dependency2;
//	}

	public String toString() {
		return "Using " + dependency1 + " and " + dependency2;
	}
}

@Component
class Dependency1 {

}

@Component
class Dependency2 {

}


@Configuration
@ComponentScan
public class DepInjectionLauncherApplication {

	public static void main(String[] args) {

		try(var context =
				new AnnotationConfigApplicationContext
				   (DepInjectionLauncherApplication.class)) {


	Arrays.stream(context.getBeanDefinitionNames()).forEach(System.out::println);

			System.out.println(context.getBean(YourBusinessClass.class));

		}
	}
}
```

Spring Framework: Key Terminology and Concepts

1. @Component

- **Definition:** Annotation placed on a class to tell Spring to manage it as a bean.
- **Effect:**
  - If the class is found during component scanning, Spring will create an instance (a bean) and manage its lifecycle.
- **Usage Example:**
  - Used on GameRunner, MarioGame, YourBusinessClass, and their dependencies in previous examples.

2. Dependency

- **Definition:**
  - An object that another object requires to function.
- **Example:**
  - GameRunner needs a GamingConsole implementation (MarioGame, SuperContraGame, or PacmanGame).
    Here, the console implementation is a dependency of GameRunner.

3. Component Scan

- **Definition:**
  - The process by which Spring searches for classes annotated with @Component (or similar annotations) to register as beans.
- **How to Use:**
  - Add @ComponentScan to a configuration class.
  - You can specify a package to scan, or if omitted, Spring scans the package of the config class and its subpackages.
- **Example:**
  - @ComponentScan("com.in28minutes") scans com.in28minutes and all its subpackages.

4. Dependency Injection (DI)

- **Definition:**
  - The process of Spring identifying the beans and their dependencies, then automatically wiring them together.
- **What Happens:**
  - Spring scans for components, finds dependencies for each, and connects them as needed.
- **Why Important:**
  - Reduces manual wiring and object creation in your code.

5. Inversion of Control (IoC)

- **Definition:**
  - The shift of control for object creation and dependency management from the programmer to the Spring framework.
- **Explanation:**
  - In traditional code, programmers explicitly create objects and wire

dependencies (`new PacmanGame()`, `new GameRunner(game)`).
    - With IoC, programmers declare what they need (via annotations/configuration), and Spring handles object creation and wiring.
- **Result:**
    - The framework, not the programmer, is "in control" of dependencies.

6. Spring Bean

- **Definition:**
    - Any object managed by the Spring framework (i.e., created and wired by Spring).
- **How Created:**
    - Typically via `@Component`, `@Service`, `@Repository`, or `@Controller`, or via explicit configuration.

7. IoC Container

- **Definition:**
    - The part of Spring responsible for managing the lifecycle of beans and their dependencies.
- **Types:**
    - **ApplicationContext:** Full-featured container (most commonly used).
    - **BeanFactory:** Simpler, less commonly used in modern Spring applications.
- **Advice:**
    - Use `ApplicationContext` for almost all real-world scenarios.

8. Autowiring

- **Definition:**
    - The process by which Spring automatically injects dependencies into beans.
- **How:**
    - Spring checks constructors, fields, or setters (depending on your annotations and injection style) and provides the required dependencies.
- **Example:**
    - If `GameRunner` has a constructor needing a `GamingConsole`, Spring will find and inject the appropriate bean automatically.

9. How These Concepts Work Together

- `@Component` marks beans for Spring to manage.
- `@ComponentScan` tells Spring where to look for beans.
- Spring's **IoC container** scans, creates, and wires beans (dependency injection).
- The process of connecting dependencies is **autowiring**.
- All managed objects are called **Spring beans**.
- The overall control of object creation shifts from programmer to framework (**Inversion of Control**).

In summary:
These core concepts—`@Component`, dependency, component scan, dependency injection, inversion of control, beans, IoC container, and autowiring—form the foundation of working with the Spring framework. Understanding them allows you to

write clean, maintainable, and loosely-coupled applications where the framework handles the complex task of managing object creation and dependencies for you.

Spring Framework: @Component vs @Bean — Differences, Usage, and Recommendations

1. Where Can You Use Each Annotation?

- **@Component:**
  - Used directly on *any* Java class (e.g., your own MarioGame, GameRunner classes).
  - Tells Spring: "Automatically create and manage a bean for this class during component scanning."
- **@Bean:**
  - Used on methods inside a *Spring configuration class* (a class annotated with @Configuration).
  - The annotated method returns an object, and Spring registers the returned object as a bean.
  - Example:
    ```
    @Bean
    public MyBean myBean() { ... }
    ```

2. Which is Easier to Use?

- **@Component is easier:**
  - Just annotate your class with @Component—no extra code required.
  - Spring handles bean creation and dependency injection for you.
- **@Bean is more complex:**
  - You write the bean creation code inside a method.
  - You are responsible for setting up dependencies (via method parameters or explicit code).

3. How Does Autowiring Work?

- **With @Component:**
  - You can use **constructor injection**, **setter injection**, or **field injection**:
    - **Constructor Injection:** Dependencies are injected via the constructor (recommended).
    - **Setter Injection:** Dependencies are injected via setter methods.
    - **Field Injection:** Dependencies are injected directly into fields (with @Autowired).
  - Example:
    ```
    @Component
    public class MyComponent {
        @Autowired
        private SomeDependency dep;
        // ...or via constructor/setter
    }
    ```

- **With @Bean:**
  - Dependencies can be injected as method parameters:
    ```
    @Bean
    public Person person(Name name, Address address) { ... }
    ```
  - Or by calling other bean methods inside the configuration class.

4. Who Creates the Beans?

- **@Component:**
  - Spring's component scanner finds all @Component-annotated classes in the scanned packages.
  - Spring itself creates and manages these beans, resolving dependencies automatically.
- **@Bean:**
  - *You* write the code to create the bean (inside a @Configuration class).
  - Spring manages the returned object as a bean, but you control its creation.

5. When Should You Use Each?

- **Use @Component for:**
  - Most of your own application classes.
  - When you are writing your own components, services, repositories, etc.
  - It's the simplest, most idiomatic approach in Spring.
- **Use @Bean for:**
  - When you need custom business logic to create a bean (e.g., conditional creation, extra setup).
  - When you need to create beans for *third-party libraries* (where you can't add @Component).
    - Example: Spring Security beans, external library classes, or objects that you do not control the source code for.

6. Summary Table

Annotation	Where to Use	Who Creates the Bean	Dependency Injection	Recommended For
@Component	On your class	Spring (auto-scan)	Constructor, setter, field	Your own app classes (services, controllers, etc)
@Bean	On method in config	You (method logic)	Method parameters or calls	Beans for 3rd-party libraries, custom setup

7. Key Takeaways

- **Prefer @Component** for your own code—just annotate and go!
- **Use @Bean** when you:
  - Need custom creation logic.
  - Are integrating with third-party libraries.
- **Both support autowiring**, but through different mechanisms.

In summary:

- Use @Component for your own classes—it's easy, clean, and fully supported by Spring's scanning and autowiring.

- Use @Bean when you need more control over bean creation, or to register beans from libraries you can't modify.
- Both approaches are fundamental in Spring, and knowing when to use each is key to effective Spring development.

Why Do We Have a Lot of Dependencies? & The Spring Solution

1. Real-World Applications Have Many Dependencies

- **Simple Apps (like GameRunner HelloWorld):**
  - Only a few classes, little to no dependency management needed.
- **Real-World Apps:**
  - Multiple layers: **Web Layer**, **Business Layer**, **Data Layer**.
  - Each layer *depends on* the layer below:
    - Web → Business → Data
  - Example:
    - A business layer class depends on a data layer class (data layer is a dependency).
  - There are **thousands** of such dependencies in large applications.

2. Spring Framework Simplifies Dependency Management

- **Without Spring:**
  - Developers must manually create, wire, and manage all object dependencies.
- **With Spring:**
  - You focus on *business logic*, not on wiring and managing objects.
  - **Spring manages the lifecycle** of your objects (beans).
  - **How?**
    - Mark classes as components with @Component.
    - Mark dependencies with @Autowired.
    - Spring will:
      - Instantiate objects
      - Inject dependencies
      - Manage the entire system for you

3. Example Exercise: BusinessCalculationService

Goal:

Create a flexible business service that can use different data sources.

Classes and Interfaces to Create:

- **DataService (Interface)**
  - int[] retrieveData();

- **MongoDbDataService (Implements DataService)**
  - Returns `new int[] { 11, 22, 33, 44, 55 };`
  - Should be marked as `@Component` and `@Primary`
- **MySQLDataService (Implements DataService)**
  - Returns `new int[] { 1, 2, 3, 4, 5 };`
  - Should be marked as `@Component`
- **BusinessCalculationService**
  - Depends on `DataService`
  - Uses constructor injection (`@Autowired` or plain constructor in recent Spring versions)
  - Method: `int findMax()`
    - Returns the maximum value from the data provided by the injected `DataService`
    - Uses `Arrays.stream(dataService.retrieveData()).max().orElse(0);`

4. Exercise Steps (What You Need to Do):

1. **Create all interfaces and classes as outlined above.**
2. **Use constructor injection** to inject `DataService` into `BusinessCalculationService`.
3. **Mark MongoDbDataService as `@Primary`** so it is chosen by default.
4. **Annotate all components with `@Component`** (and `@Primary` where needed).
5. **Create a Spring context** (using `AnnotationConfigApplicationContext` and a configuration class).
6. **Retrieve the `BusinessCalculationService` bean** from the context.
7. **Call the `findMax()` method** and verify it returns the correct value.

5. Why Use @Primary?

- If multiple beans implement the same interface (`DataService`), Spring doesn't know which one to inject.
- Marking `MongoDbDataService` as `@Primary` tells Spring to use it by default when injecting `DataService`.

6. Key Spring Annotations Used

- `@Component` – marks a class as a Spring-managed bean/component.
- `@Primary` – gives preference to a bean when multiple candidates are available for injection.
- `@Autowired` – marks a dependency to be injected (often not needed on constructors in recent Spring versions).

7. Key Takeaways

- **Spring makes managing dependencies easy**—you don't need to manually wire classes.
- **Focus on business logic, not plumbing**—Spring handles object creation, dependency injection, and lifecycle.
- Use `@Component` and `@Autowired` to let Spring do its work.
- Use `@Primary` to resolve ambiguity when you have multiple beans of the same type.
- Real-world Spring applications scale to thousands of dependencies thanks to this powerful mechanism.

In summary:
Spring allows you to focus on your application's business logic by handling the complex task of dependency management for you. By using annotations like @Component, @Primary, and @Autowired, you can build flexible, maintainable, and scalable applications with ease.

RealWorldSpringContextLauncherApplication.java

```java
package com.in28minutes.learn_spring_framework.examples.c1;

import java.util.Arrays;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan
public class RealWorldSpringContextLauncherApplication {

	public static void main(String[] args) {

		try(var context =
				new AnnotationConfigApplicationContext
				   (RealWorldSpringContextLauncherApplication.class)) {

			Arrays.stream(context.getBeanDefinitionNames())
				.forEach(System.out::println);

			System.out.print(

	context.getBean(BusinessCalculationService.class).findMax());
		}
	}
}
```

DataService.java

```java
package com.in28minutes.learn_spring_framework.examples.c1;

public interface DataService {
	int[] retrieveData();
}
```

MongoDbDataService.java

```java
package com.in28minutes.learn_spring_framework.examples.c1;


import org.springframework.context.annotation.Primary;
import org.springframework.stereotype.Component;

@Component
@Primary
public class implements DataService{

	@Override
	public int[] retrieveData() {
		return new int[] { 11, 22, 33, 44, 55 };
	}
```

```
}
```

MySQLDataService.java

```
package com.in28minutes.learn_spring_framework.examples.c1;


import org.springframework.stereotype.Component;

@Component
public class implements DataService{

	@Override
	public int[] retrieveData() {
		return new int[] { 1, 2, 3, 4, 5 };
	}

}
```

BusinessCalculationService.java

```
package com.in28minutes.learn_spring_framework.examples.c1;

import java.util.Arrays;

import org.springframework.stereotype.Component;

@Component
public class BusinessCalculationService {

	private DataService dataService;

	public BusinessCalculationService(DataService dataService) {
		super();
		this.dataService = dataService;
	}

	public int findMax() {
		return Arrays.stream(dataService.retrieveData()).max().orElse(0);
	}
```

Spring Framework Section Recap & What's Next: Key Concepts and Upcoming Topics

1. Recap: What You've Learned So Far

- **Introduction to Spring Framework:**
  - Gained foundational understanding of the Spring framework and its purpose.
- **Coupling and Loose Coupling:**
  - Learned what *coupling* is in software design.
  - Explored how to achieve *loose coupling* using Java interfaces, making code more flexible and maintainable.
- **Launching the Spring Container:**
  - Understood how to start a Spring container using an *application context*.

    - Created and managed Spring beans within the container.
- **Java Bean vs. Spring Bean:**
    - Clarified the difference between a plain Java bean and a Spring-managed bean.
- **Core Spring Concepts:**
    - Understood what *dependencies* are in applications.
    - Learned about *autowiring* and *dependency injection*—and their different types.
- **Essential Spring Annotations:**
    - Explored the use and meaning of:
        - @Bean
        - @Component
        - @Configuration
        - @Primary
        - @Qualifier
    - Learned when and how to use each annotation.
- **Hands-on Learning:**
    - All concepts were learned through practical, hands-on coding and exercises.

2. Importance of These Concepts

- Mastering Spring fundamentals not only helps you build robust Spring applications, but also makes debugging much easier.
- Solid understanding of these basics sets you on the path to becoming a *great developer*.

3. What's Coming Next? (Sneak Peek)

- **Advanced Spring Topics:**
    - **Lazy Initialization:**
        - Learn how to delay bean initialization until it's actually needed (not at container startup).
    - **Bean Scopes:**
        - Dive into different scopes, focusing on *Singleton* and *Prototype* beans.
    - **Lifecycle Callbacks:**
        - Understand @PostConstruct (actions after bean is ready) and @PreDestroy (actions before bean is destroyed).
    - **Java EE Evolution:**
        - Brief history of J2EE, Java EE, Jakarta EE.
        - Learn about CDI (Context and Dependency Injection), a key Jakarta EE specification.
    - **Configuration Styles:**
        - Explore XML-based configuration vs Java-based configuration in Spring.
        - Learn when to use each approach.
    - **Stereotype Annotations:**
        - Go beyond @Component to understand @Service, @Repository, and other stereotypes.
    - **Big Picture:**
        - Overview of Spring modules and projects.
        - Understand why Spring is the leading framework in the Java world.

4. Encouragement & Next Steps

- You've built a strong foundation—be excited for the upcoming advanced topics!
- The next section will deepen your understanding of Spring's power and versatility.
- Let's continue the journey and become even more proficient with the Spring framework.

In summary:
You've learned core Spring concepts—dependency injection, bean management, and key annotations. Next, you'll explore advanced features such as lazy initialization, bean scopes, lifecycle hooks, alternative configurations, stereotype annotations, and the broader Spring ecosystem. Get ready to elevate your Spring skills even further!

Congratulations & The Power of Consistent Learning

1. Celebrating Your Progress

- You've reached the **second milestone** of the course—well done!
- Take a moment to acknowledge your progress and commitment.

2. When Do You Learn and Retain the Most?

- **Consistent learning** yields the best results:
  - Spending a few hours *every day* over a week or month leads to deeper understanding and better retention.
  - Cramming everything into a single weekend may help short-term recall, but you're less likely to remember the material in the long run.
- **Practical insight:**
  - Using what you learn regularly ("day in, day out") helps cement your knowledge.

3. How to Build Consistency in Your Learning?

A. Create a Learning Plan

- Sit down and **write out a specific plan** for the next two weeks.
- Be detailed:
  - Specify which lecture numbers or topics you'll cover each day.
  - Example:
    - Day 1: Lectures 10–18
    - Day 2: Lectures 19–25
    - Day 3: Lectures 26–33

B. Make Your Plan Visible

- Place your plan **somewhere you can see every day** (e.g., near your desk).
- **Visibility increases your chances of success** by over 50%.

- It serves as a daily reminder and motivator.

4. Extra Motivation

- Consider sharing your learning plan in the course Q&A/community.
  - This adds accountability and may inspire others.

5. Key Takeaways

Advice	Why It Matters
Learn a little every day	Builds long-term understanding and memory
Create a clear plan	Gives you direction and focus
Make it visible	Increases your commitment and follow-through
Use the material often	Reinforces and deepens your knowledge

6. Encouragement

- Consistent, well-planned learning is the best way to master new skills and remember them long-term.
- Keep up your momentum—your progress so far is proof that you can do it!

In summary:
You learn and remember best by studying consistently over time, not by cramming. To achieve this, create a specific, visible learning plan and stick to it. Celebrate your progress, and keep going—you're on the right track!

Exploring Spring Framework Advanced Features

17 July 2025 03:53

Exploring Lazy Initialization of Spring Beans

1. Default Behavior: Eager Initialization

- **By default, Spring beans are eagerly initialized:**
  - All beans are created and their dependencies injected as soon as the Spring context starts.
  - This means *all initialization logic* (such as constructor code) runs at startup, even if you never use the bean.

Example from Code:

```
@Component
class ClassA {}

@Component
class ClassB {
    public ClassB(ClassA classA) {
        System.out.println("Some Initialization logic");
        this.classA = classA;
    }
}
```

- If you launch the context, "Some Initialization logic" is printed **immediately**, even if you never access ClassB.

2. What is Lazy Initialization?

- **Lazy Initialization** delays the creation of a bean until it is actually needed (i.e., when it is first requested from the context).
- You enable this by annotating the bean with @Lazy.

Your Code Example:

```
@Component
@Lazy
class ClassB {
    public ClassB(ClassA classA) {
        System.out.println("Some Initialization logic");
        this.classA = classA;
    }

    public void doSomething() {
        System.out.println("Do Something");
```

```
    }
}
```

- With @Lazy, ClassB is **not initialized at startup.**
- It is only initialized when you explicitly request it:
  context.getBean(ClassB.**class**).doSomething();

3. Demonstration (What Happens When You Run the App)

- When the Spring context starts:
  - You see "Intitialization of context is completed" printed.
  - **No initialization logic from ClassB yet!**
- When you request ClassB from the context:
  - Only then you see "Some Initialization logic" (constructor runs).
  - Then "Do Something" is printed.

4. How to Use @Lazy

- **On a class:**
  - Add @Lazy to a @Component class.
- **On a @Bean method:**
  - Add @Lazy to a method in a @Configuration class.
- **On a @Configuration class:**
  - All beans defined in that configuration become lazy-initialized.

5. When Should You Use Lazy Initialization?

- **Eager initialization is recommended** for most cases.
  - **Why?**
    - Any errors in configuration or bean setup are detected immediately at startup.
    - You avoid runtime surprises.
- **Lazy initialization is NOT recommended and not frequently used.**
  - **Why?**
    - Errors in bean configuration are only discovered at runtime, when the bean is first used.
    - This can lead to harder-to-debug, delayed failures.
- **When to consider lazy?**
  - If a bean has very expensive or complex initialization, and you do not want to delay application startup.
  - For rarely-used, optional, or heavy beans.

6. How Does Spring Handle Lazy Beans?

- For lazy beans, Spring injects a *lazy-resolution proxy* at injection points.
  - The real object is only created when it is first accessed.

7. Slide Summary

Aspect	Eager Initialization	Lazy Initialization

When beans are created	At context startup	When first requested/used
Error discovery	At startup	At runtime (when bean is first used)
Usage	Default, recommended	Not recommended, rare use-cases
How to enable	(No extra annotation)	@Lazy on class, method, or config
Proxy	Not used	Lazy proxy injected

8. Key Takeaways

- **Spring beans are eager by default.**
- Use @Lazy only for special scenarios (complex, rarely-used beans).
- Lazy beans can be defined with @Lazy on classes, bean methods, or even the config class.
- Prefer eager initialization for early error detection and a predictable application lifecycle.

In summary:
Eager initialization is the default and preferred approach in Spring—beans are created at startup, ensuring early error detection. Use @Lazy sparingly, for beans with expensive initialization or special requirements, understanding that errors may only appear at runtime.

```
package com.in28minutes.learn_spring_framework.examples.d1;

import java.util.Arrays;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Lazy;
import org.springframework.stereotype.Component;

@Component
class ClassA {

}

@Component
@Lazy
class ClassB {

	private ClassA classA;

	public ClassB(ClassA classA) {
		// Logic
		System.out.println("Some Initialization logic");
		this.classA = classA;
	}

	public void doSomething() {
		System.out.println("Do Something");
	}
}

@Configuration
@ComponentScan
```

```
public class LazyInitializationLauncherApplication {

	public static void main(String[] args) {

		try(var context =
				new AnnotationConfigApplicationContext
				   (LazyInitializationLauncherApplication.class)) {

			System.out.println("Intitialization of context is completed");

			context.getBean(ClassB.class).doSomething();

		}
	}
}
```

Lazy Initialization vs Eager Initialization in Spring

1. Initialization Time

- **Lazy Initialization:**
  - **When?** Bean is created only when it is first needed (i.e., when it's first requested/used in the application).
- **Eager Initialization:**
  - **When?** Bean is created as soon as the Spring context starts (at application startup).

2. Default Behavior

- **Lazy Initialization:**
  - **Not default** in Spring.
  - Must be explicitly enabled with `@Lazy` or `@Lazy(value = true)`.
- **Eager Initialization:**
  - **Default** behavior in Spring.
  - Achieved by not using `@Lazy` (or using `@Lazy(value = false)`).

3. Enabling Each Mode (Code Snippets)

- **Lazy Initialization:**

```
@Component
@Lazy
class MyBean { ... }
// or
@Lazy(value = true)
```

- **Eager Initialization:**

```
@Component
// (No @Lazy annotation) OR
@Lazy(value = false)
class MyBean { ... }
```

4. Error Handling

- **Lazy Initialization:**
  - Errors in bean creation are only discovered at runtime, when the bean is first requested.
  - Can lead to runtime exceptions *during application usage*.
- **Eager Initialization:**
  - Errors are discovered immediately at application startup.
  - If there's a problem, the application will fail to start, making issues easier to catch and fix early.

5. Usage Frequency

- **Lazy Initialization:**
  - **Rarely used** in typical applications.
- **Eager Initialization:**
  - **Most common** and recommended approach for the vast majority of beans.

6. Memory Consumption

- **Lazy Initialization:**
  - Potentially uses less memory at startup, since beans aren't created until needed.
- **Eager Initialization:**
  - All beans are created and loaded into memory at startup.

7. Recommended Scenarios

Mode	When to Use
Lazy	For beans that are **rarely used** or expensive to create and not needed at startup
Eager (default)	For **most beans**; ensures early error detection and predictable application startup

8. Slide Comparison Table

Aspect	Lazy Initialization	Eager Initialization
Initialization time	When bean is first used	At application startup
Default	Not default	Default
Code Snippet	@Lazy or @Lazy(value=true)	@Lazy(value=false) or no @Lazy
Error Handling	Errors at runtime (when bean is requested)	Errors at startup (prevents startup)
Usage	Rarely used	Very frequently used
Memory Consumption	Less (until bean is initialized)	All beans loaded at startup
Recommended	Rarely used beans	Most beans

Scenario		

9. Key Takeaways

- **Eager initialization is the default and recommended for most situations.**
- **Lazy initialization** is useful for special cases: beans that are rarely used or expensive to create.
- **Error detection is earlier and safer with eager initialization.**
- **Memory usage can be lower with lazy beans—but only if those beans are not used often.**

In summary:
Use eager initialization (the default) for almost all your beans—it's safer and more predictable. Use lazy initialization only for rare, special cases where beans are seldom needed and you want to optimize startup time or memory.

Spring Bean Scopes: Singleton vs Prototype (and More)

1. What are Bean Scopes in Spring?

- **Bean scope** defines **how many instances** of a bean Spring creates and how long those instances live.
- Beans can be configured to have different scopes depending on the use case.

2. Most Common Scopes

A. Singleton (Default)

- **Definition:** Only **one object instance per Spring IoC container.**
- **Behavior:** Every time you ask for the bean, you get the *same instance*.
- **How to use:** Just annotate with `@Component` (no extra scope annotation needed).
- **Example from code:**
  ```
  @Component
  class NormalClass {}
  ```
- **Demo Output:**
  All calls to `context.getBean(NormalClass.class)` print the *same* object reference.

B. Prototype

- **Definition: Multiple object instances per Spring IoC container.**
- **Behavior:** Every time you ask for the bean, you get a *new instance*.
- **How to use:** Annotate with @Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE) and @Component.
- ConfigurableBeanFactory.SCOPE_PROTOTYPE is nothing but "prototype".
- **Example from code:**

```
@Scope(value = ConfigurableBeanFactory.SCOPE_PROTOTYPE)
@Component
class PrototypeClass {}
```

- **Demo Output:**
  Each call to context.getBean(PrototypeClass.class) prints a *different* object reference.

3. Demo Code and Output Recap

```
System.out.println(context.getBean(NormalClass.class)); // same instance each time
System.out.println(context.getBean(PrototypeClass.class)); // new instance each time
```

Sample Output:

```
NormalClass@408d971b
NormalClass@408d971b
NormalClass@408d971b
NormalClass@408d971b
NormalClass@408d971b
NormalClass@408d971b
PrototypeClass@6c6cb480
PrototypeClass@3c46e67a
PrototypeClass@c730b35
PrototypeClass@206a70ef
```

- All NormalClass instances have the *same* hash (singleton).
- All PrototypeClass instances have *different* hashes (prototype).

4. Other Scopes (Web-aware Contexts Only)

- **Request:** One object per *single HTTP request*.
- **Session:** One object per *user HTTP session*.
- **Application:** One object per *web application runtime*.
- **Websocket:** One object per *WebSocket connection*.
- *Note:* These are only available in web-aware Spring ApplicationContexts.

5. Java Singleton (GOF) vs Spring Singleton

GOF – Gang Of Four

Type	Scope/Meaning
Spring Singleton	One object per Spring IoC container
Java Singleton	(GOF Pattern) One object per JVM (Java process)

- Usually, these are the same in most applications (since you typically have one

Spring IoC container per JVM), but technically, if you have multiple IoC containers in one JVM, each container would have its own "singleton" bean instance.

6. Summary Table (from Instructor's Slide)

Scope	Description
Singleton	One instance per Spring IoC container
Prototype	Many instances per Spring IoC container
Request*	One per HTTP request (web only)
Session*	One per HTTP session (web only)
Application*	One per web application runtime (web only)
Websocket*	One per WebSocket instance (web only)

** Available only in web-aware contexts.*

7. Key Takeaways

- **Singleton** is the default and most common bean scope in Spring.
- **Prototype** is used when you need a new instance each time you request a bean.
- For most use cases, use singleton unless you have a specific requirement for multiple instances.
- Web-specific scopes help manage beans tied to HTTP requests, sessions, or websockets.
- Be aware of the distinction between **Spring Singleton** (per container) and **Java Singleton** (per JVM).

In summary:

- By default, Spring beans are singletons: you always get the same instance from the container.
- Mark a bean as @Scope("prototype") to get a new instance each time.
- Web scopes allow for finer-grained bean lifecycles in web applications.
- Understand the difference between Spring's singleton and the classic Java singleton pattern.

```
package com.in28minutes.learn_spring_framework.examples.e1;

import java.util.Arrays;

import org.springframework.beans.factory.config.ConfigurableBeanFactory;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;


@Component
class NormalClass {

}

@Scope(value=ConfigurableBeanFactory.SCOPE_PROTOTYPE)
```

```
@Component
class PrototypeClass {

}

@Configuration
@ComponentScan
public class BeanScopesLauncherApplication {

	public static void main(String[] args) {

		try(var context =
				new AnnotationConfigApplicationContext
				   (BeanScopesLauncherApplication.class)) {

			System.out.println(context.getBean(NormalClass.class));
			System.out.println(context.getBean(NormalClass.class));
			System.out.println(context.getBean(NormalClass.class));

			System.out.println(context.getBean(NormalClass.class));

			System.out.println(context.getBean(NormalClass.class));

			System.out.println(context.getBean(NormalClass.class));


			System.out.println(context.getBean(PrototypeClass.class));

			System.out.println(context.getBean(PrototypeClass.class));

			System.out.println(context.getBean(PrototypeClass.class));
			System.out.println(context.getBean(PrototypeClass.class));




		}
	}
}
```

Output:

```
com.in28minutes.learn_spring_framework.examples.e1.NormalClass@408d971b
com.in28minutes.learn_spring_framework.examples.e1.NormalClass@408d971b
com.in28minutes.learn_spring_framework.examples.e1.NormalClass@408d971b
com.in28minutes.learn_spring_framework.examples.e1.NormalClass@408d971b
com.in28minutes.learn_spring_framework.examples.e1.NormalClass@408d971b
com.in28minutes.learn_spring_framework.examples.e1.NormalClass@408d971b
com.in28minutes.learn_spring_framework.examples.e1.PrototypeClass@6c6cb480
com.in28minutes.learn_spring_framework.examples.e1.PrototypeClass@3c46e67a
com.in28minutes.learn_spring_framework.examples.e1.PrototypeClass@c730b35
com.in28minutes.learn_spring_framework.examples.e1.PrototypeClass@206a70ef
```

Prototype vs Singleton Bean Scope in Spring

1. Number of Instances Created

- **Prototype Scope:**
  - **Possibly many instances** per Spring IoC container.
  - Every time you request the bean, **a new instance** is created and returned.
- **Singleton Scope:**
  - **One instance per Spring IoC container.**
  - Every time you request the bean, **the same instance** is returned (reused).

2. Default Behavior

- **Prototype:**
  - **NOT the default.** You must explicitly specify this scope.
- **Singleton:**
  - **Default scope** in Spring. If you don't specify a scope, beans are singletons.

3. How to Specify Each Scope

- **Prototype Example:**

```
@Scope(value = ConfigurableBeanFactory.SCOPE_PROTOTYPE)
@Component
class MyPrototypeBean { ... }
```

- **Singleton Example:**

```
// Either explicitly:
@Scope(value = ConfigurableBeanFactory.SCOPE_SINGLETON)
@Component
class MySingletonBean { ... }

// Or just omit @Scope (default is singleton)
@Component
class MySingletonBean { ... }
```

4. Usage Frequency

- **Prototype:**
  - **Rarely used.**
- **Singleton:**
  - **Very frequently used.**
  - The vast majority (99.99%) of Spring beans you'll create are singletons.

5. Recommended Scenarios

Scope	Recommended For	Description
Prototype	**Stateful beans**	Beans that hold user/session-specific information.
Singleton	**Stateless beans**	Beans that are generic, shared, and reusable.

- **Prototype:**
  - Useful if you need a separate bean for each user/session/operation (stateful).
- **Singleton:**
  - Best when beans are stateless and can be shared across the application.

6. Slide Summary Table

Aspect	Prototype	Singleton
Instances	Possibly Many per Spring IoC container	One per Spring IoC container
Beans	New bean instance each time requested	Same instance reused
Default	Not default	Default
Code Snippet	@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)	@Scope(ConfigurableBeanFactory.SCOPE_SINGLETON) or default
Usage	Rarely used	Very frequently used
Recommended Scenario	Stateful beans	Stateless beans

7. Key Takeaways

- **Singleton is the default and should be used for most beans, especially stateless ones.**
- **Prototype is rarely needed**—use only if you require a new instance for each request (typically for stateful beans).
- Choosing the right scope is important for correct application behavior and resource management.

In summary:

- **Singleton:** One shared instance per Spring container—default, best for stateless beans.
- **Prototype:** New instance on each request—rarely used, best for stateful beans.

Spring Bean Lifecycle: Using @PostConstruct and @PreDestroy

1. Why Use @PostConstruct and @PreDestroy?

- **@PostConstruct:**
  Lets you run **initialization logic** *after* all dependencies have been injected, but *before* the bean is used by anyone else in the application.
- **@PreDestroy:**

Lets you run **cleanup logic** *right before* the bean is removed from the Spring context (for example, when the application shuts down).

2. How Does It Work? (Code Explanation)

a. Bean with Dependencies and Lifecycle Methods

```
@Component
class SomeClass {
    private SomeDependency someDependency;

    public SomeClass(SomeDependency someDependency) {
        this.someDependency = someDependency;
        System.out.println("All dependencies are ready!");
    }

    @PostConstruct
    public void initialize() {
        someDependency.getReady();
    }

    @PreDestroy
    public void cleanup() {
        System.out.println("Clean Up");
    }
}
```

- **Constructor**: Runs first, after dependencies are injected by Spring. Prints "All dependencies are ready!".
- **@PostConstruct initialize()**: Runs immediately after the constructor. Here, you can perform additional setup using injected dependencies.
  Calls someDependency.getReady().
- **@PreDestroy cleanup()**: Runs *just before* the bean is destroyed (e.g., when the application context is closed). Useful for releasing resources or closing connections.

b. Dependency Bean

```
@Component
class SomeDependency {
    public void getReady() {
        System.out.println("Some logic using SomeDependency");
    }
}
```

3. Application Setup and Output

```
@Configuration
@ComponentScan
public class PrePostAnnotationContextLauncherApplication {
    public static void main(String[] args) {
        try(var context = new
AnnotationConfigApplicationContext(PrePostAnnotationContextLauncherApplication.cla
ss)) {
            Arrays.stream(context.getBeanDefinitionNames()).forEach(System.out::println);
```

```
    }
  }
}
```

Output Explanation:
All dependencies are ready!
Some logic using SomeDependency
... (Spring's internal beans and user beans printed) ...
Clean Up

- **"All dependencies are ready!"**: Constructor of SomeClass runs.
- **"Some logic using SomeDependency"**: @PostConstruct method runs, calling the dependency method.
- **Bean names are listed.**
- **"Clean Up"**: @PreDestroy method runs when context is closed (end of try block).

4. When and Why to Use These Annotations

Annotation	When is it called?	Typical Usage
@PostConstruct	After dependencies are injected, before use	Initialization, setup, fetching initial data
@PreDestroy	Before bean is destroyed/context is closed	Cleanup, closing connections, releasing resources

- **@PostConstruct** is often used to perform actions that require all dependencies to be ready (e.g., open a connection, load data).
- **@PreDestroy** is typically used for cleanup (e.g., closing a database connection, freeing resources).

5. Summary Table

Step	What Happens
Bean Instantiated	Constructor runs, dependencies injected
@PostConstruct Method	Runs after constructor, does post-initialization
Bean Used in Application	...
Application Context Closes	@PreDestroy method runs, bean cleaned up

6. Key Takeaways

- Use @PostConstruct for logic that should run after all dependencies are injected, but before the bean is available for use.
- Use @PreDestroy for cleanup logic before the bean is removed from the context.
- Both annotations help manage the **lifecycle of beans** in a clean, declarative way.

In summary:

- @PostConstruct and @PreDestroy are powerful lifecycle hooks for managing initialization and cleanup in your Spring beans.
- They ensure your beans are always properly prepared and cleaned up, reducing resource leaks and setup errors.

```
package com.in28minutes.learn_spring_framework.examples.f1;
```

```
import java.util.Arrays;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.stereotype.Component;

import jakarta.annotation.PostConstruct;
import jakarta.annotation.PreDestroy;


@Component
class SomeClass {
	private SomeDependency someDependency;

	public SomeClass(SomeDependency someDependency) {
		super();
		this.someDependency = someDependency;
		System.out.println("All dependencies are ready!");
	}

	@PostConstruct
	public void initialize() {
		someDependency.getReady();
	}

	@PreDestroy
	public void cleanup() {
		System.out.println("Clean Up");
	}

}

@Component
class SomeDependency {

	public void getReady() {
		System.out.println("Some logic using SomeDependency");
	}

}

@Configuration
@ComponentScan
public class PrePostAnnotationContextLauncherApplication {

	public static void main(String[] args) {

		try(var context =
				new AnnotationConfigApplicationContext
				   (PrePostAnnotationContextLauncherApplication.class)) {


	Arrays.stream(context.getBeanDefinitionNames()).forEach(System.out::println);
		}
	}
}
```

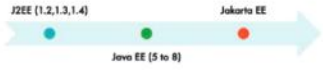
Output:

```
All dependencies are ready!
Some logic using SomeDependency
org.springframework.context.annotation.internalConfigurationAnnotationProcessor
org.springframework.context.annotation.internalAutowiredAnnotationProcessor
org.springframework.context.annotation.internalCommonAnnotationProcessor
org.springframework.context.event.internalEventListenerProcessor
org.springframework.context.event.internalEventListenerFactory
prePostAnnotationContextLauncherApplication
someClass
someDependency
Clean Up
```

The Evolution of J2EE → Java EE → Jakarta EE

Evolution of Jakarta EE: vs J2EE vs Java EE

- Enterprise capabilities were **initially built into JDK**
- With time, they were separated out:
  - **J2EE** - Java 2 Platform Enterprise Edition
  - **Java EE** - Java Platform Enterprise Edition (Rebranding)
  - **Jakarta EE** (Oracle gave Java EE rights to the Eclipse Foundation)
    - **Important Specifications**:
      - Jakarta Server Pages (JSP)
      - Jakarta Standard Tag Library (JSTL)
      - Jakarta Enterprise Beans (EJB)
      - Jakarta RESTful Web Services (JAX-RS)
      - Jakarta Bean Validation
      - Jakarta Contexts and Dependency Injection (CDI)
      - Jakarta Persistence (JPA)
    - Supported by **Spring 6 and Spring Boot 3**
      - That's why we use jakarta. *packages (instead of javax.)*



1. What are J2EE, Java EE, and Jakarta EE?

- **Enterprise features were initially part of the JDK** (Java Development Kit).
- Over time, these features were **separated out** to better modularize the platform

and allow for innovation.

Name Evolution:

- **J2EE (Java 2 Platform, Enterprise Edition):**
  - The original umbrella for enterprise Java technologies (e.g., servlets, JSP, EJB).
- **Java EE (Java Platform, Enterprise Edition):**
  - A rebranding of J2EE, with the same purpose.
- **Jakarta EE:**
  - Oracle transferred Java EE to the **Eclipse Foundation**.
  - The community voted to rename it to **Jakarta EE**.

Key Point:

All three—**J2EE**, **Java EE**, and **Jakarta EE**—refer to the **same family of enterprise Java standards**, just at different points in history.

2. Important Specifications in Jakarta EE

- **Jakarta Server Pages (JSP):**
  - Used for rendering web page views; formerly Java Server Pages.
- **Jakarta Standard Tag Library (JSTL):**
  - Provides tags for dynamic web content; formerly Java Standard Tag Library.
- **Jakarta Enterprise Beans (EJB):**
  - For building scalable, transactional, multi-user secure enterprise-level applications.
- **Jakarta RESTful Web Services (JAX-RS):**
  - Specification for building REST APIs.
- **Jakarta Bean Validation:**
  - Standard for validating bean properties.
- **Jakarta Contexts and Dependency Injection (CDI):**
  - Specification for dependency injection in Jakarta EE.
- **Jakarta Persistence (JPA):**
  - API for interacting with relational databases.

3. Why is it called "Jakarta" now?

- **Oracle** was the owner of Java EE.
- Oracle **transferred the rights** to the **Eclipse Foundation**.
- The name was changed to avoid trademark issues and to mark a new era of community-driven development.

4. Relationship to Spring

- **Spring Framework 6** and **Spring Boot 3** fully support Jakarta EE.
  - That's why you see `jakarta.*` packages in modern Spring applications (instead of the old `javax.*`).
- Many annotations and APIs in Spring now import from `jakarta.*`.

5. Key Takeaways

Term	Meaning & Evolution
J2EE	Java 2 Platform, Enterprise Edition (original name)

Java EE	Java Platform, Enterprise Edition (rebranding)
Jakarta EE	Community-driven version, now managed by Eclipse Foundation

- **All are a group of specifications** (not implementations!):
  - JSP, JSTL, EJB, JAX-RS, Bean Validation, CDI, JPA, etc.
- **Modern Spring uses and supports Jakarta EE standards.**

6. Why does this matter to you?

- When you see jakarta.* in Spring code, it means you're using the latest, standards-based APIs that are supported by both Jakarta EE and Spring.
- Knowing this history helps you understand the evolution and interoperability of enterprise Java technologies.

In summary:
J2EE (now Jakarta EE) is a set of enterprise Java standards that has evolved through rebranding and stewardship changes. Today, Spring Framework 6 and Spring Boot 3 use and support Jakarta EE specifications, which is why you use jakarta.* packages in your modern Spring applications.

Jakarta Contexts and Dependency Injection (CDI) & Spring

Make changes in pom.xml:

```
<dependency>
     <groupId>jakarta.inject</groupId>
     <artifactId>jakarta.inject-api</artifactId>
     <scope>2.0.1</scope>
</dependency>
Add the above before:
<dependency>
     <groupId>org.springframework.boot</groupId>
     <artifactId>spring-boot-starter-test</artifactId>
     <scope>test</scope>
</dependency>
```

1. What is CDI?

- **CDI** stands for **Contexts and Dependency Injection.**
- It is a **specification** (a set of interfaces and annotations), originally introduced in Java EE 6 (2009), now part of **Jakarta EE.**
- **Spring Framework** (since v1, 2004) also implements CDI concepts and supports CDI annotations.

2. CDI vs Spring: Key Annotations

CDI Annotation	Spring Equivalent	Purpose
@Inject	@Autowired	Injects dependencies (constructor, setter, field)
@Named	@Component	Declares a managed bean/component
@Qualifier	@Qualifier	Disambiguates which bean to inject
@Scope	@Scope	Sets bean scope (singleton, prototype, etc.)

@Singleton	@Scope("singleton")	Declares singleton scope

- **You can use CDI annotations in a Spring application**—Spring recognizes and supports them.

3. Example: Using CDI in a Spring Application

a. BusinessService Bean

```
//@Component
@Named  // CDI annotation instead of @Component
class BusinessService {
   private DataService dataService;

   // @Autowired
   @Inject  // CDI annotation instead of @Autowired
   public void setDataService(DataService dataService) {
      System.out.println("Setter Injection");
      this.dataService = dataService;
   }

   public DataService getDataService() {
      return dataService;
   }
}
```

b. DataService Bean

```
//@Component
@Named  // CDI annotation instead of @Component
class DataService {}
```

c. Application Launcher

```
@Configuration
@ComponentScan
public class CdiContextLauncherApplication {
   public static void main(String[] args) {
      try(var context = new
AnnotationConfigApplicationContext(CdiContextLauncherApplication.class)) {
         Arrays.stream(context.getBeanDefinitionNames()).forEach(System.out::println);
         System.out.println(context.getBean(BusinessService.class).getDataService());
      }
   }
}
```

d. Output

```
Setter Injection
...
businessService
dataService
com.in28minutes.learn_spring_framework.examples.g1.DataService@7817fd62
```

- **"Setter Injection"**: Confirms that the setter method was called and the dependency was injected using CDI's @Inject.

4. Key Takeaways from the Lecture & Slides

- **CDI is just a specification** (not an implementation); Spring provides support for CDI annotations.
- CDI and Spring annotations are **interchangeable** for the common cases (e.g., @Inject ≈ @Autowired, @Named ≈ @Component).
- You **do not need to use CDI annotations in Spring** (Spring's own annotations are sufficient), but **it's good to be aware** of them for interoperability and understanding legacy or standards-based code.
- Modern Spring applications (especially with Jakarta EE) often use jakarta.inject.* packages due to Spring's support for Jakarta EE 9+.

5. Summary Table

Feature	Spring Annotation	CDI Annotation (Jakarta)	Description
Dependency Inject	@Autowired	@Inject	Injects dependencies
Component	@Component	@Named	Declares a managed bean
Qualifier	@Qualifier	@Qualifier	Qualifies which bean to inject
Scope	@Scope	@Scope, @Singleton	Sets bean scope

6. Why is This Useful?

- Knowing CDI allows you to:
  - Understand standards-based Java EE/Jakarta EE projects.
  - Integrate or migrate between Spring and Jakarta EE platforms.
  - Use standard annotations in Spring for interoperability or preference.

In summary:
CDI (Contexts and Dependency Injection) is a Java/Jakarta EE specification that defines standard annotations for dependency injection. Spring fully supports these annotations, so you can use @Inject and @Named as alternatives to Spring's @Autowired and @Component. This enhances compatibility and flexibility in modern Java enterprise development.

```
package com.in28minutes.learn_spring_framework.examples.g1;

import java.util.Arrays;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.stereotype.Component;

import jakarta.inject.Inject;
import jakarta.inject.Named;

//@Component
@Named
class BusinessService {
```

```
	private DataService dataService;

//	@Autowired
	@Inject
	public void setDataService(DataService dataService) {
		System.out.println("Setter Injection");
		this.dataService = dataService;
	}

	public DataService getDataService() {
		return dataService;
	}
}

//@Component
@Named
class DataService {

}

@Configuration
@ComponentScan
public class CdiContextLauncherApplication {

	public static void main(String[] args) {

		try(var context =
				new AnnotationConfigApplicationContext
				   (CdiContextLauncherApplication.class)) {


	Arrays.stream(context.getBeanDefinitionNames()).forEach(System.out::println);

				System.out.println(context.getBean(BusinessService.class)
						.getDataService());
		}
	}
}
```

```
Setter Injection
org.springframework.context.annotation.internalConfigurationAnnotationProcessor
org.springframework.context.annotation.internalAutowiredAnnotationProcessor
org.springframework.context.annotation.internalCommonAnnotationProcessor
org.springframework.context.event.internalEventListenerProcessor
org.springframework.context.event.internalEventListenerFactory
cdiContextLauncherApplication
businessService
dataService
com.in28minutes.learn_spring_framework.examples.g1.DataService@7817fd62
```

Spring XML Configuration: Defining Beans the Old-School Way

1. What is Spring XML Configuration?

- **XML configuration** was the original way to configure beans and dependencies in Spring applications.
- All beans, wiring, and even component scanning were specified using XML files.
- With modern Spring (Spring 3+), **Java-based configuration** and annotations are now the standard, but understanding XML is important for maintaining legacy projects.

2. Your Example: contextConfiguration.xml

```
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
   xsi:schemaLocation="
      http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

<!-- Simple beans with values -->
   <bean id="name" class="java.lang.String">
      <constructor-arg value="Nilesh" />
   </bean>
   <bean id="age" class="java.lang.String">
      <constructor-arg value="24" />
   </bean>

<!-- (Optional) Component scan for annotation-based beans -->
   <!-- <context:component-scan base-
package="com.in28minutes.learn_spring_framework.game" /> -->

<!-- Explicit bean definitions for application classes -->
   <bean id="game"
class="com.in28minutes.learn_spring_framework.game.PacmanGame" />
   <bean id="gameRunner"
class="com.in28minutes.learn_spring_framework.game.GameRunner">
      <constructor-arg ref="game" />
   </bean>
</beans>
```

Key Points:

- **Primitive beans:** You can define beans for primitive types or any class, like `java.lang.String`.
- **Bean wiring:** Use `<constructor-arg>` with `value` for primitives or `ref` to inject other beans.
- **Component scan:** Can enable scanning for `@Component` classes (commented out in your example).
- **Explicit beans:** Manually wire up your main application beans (e.g., `PacmanGame`, `GameRunner`).

3. Java Code to Load XML Config

```
public class XmlConfigurarionContextLauncherApplication {
    public static void main(String[] args) {
        try(var context = new
ClassPathXmlApplicationContext("contextConfiguration.xml")) {
            Arrays.stream(context.getBeanDefinitionNames()).forEach(System.out::println);
            System.out.println(context.getBean("name"));
            System.out.println(context.getBean("age"));
            context.getBean(GameRunner.class).run();
        }
    }
}
```

How it Works:

- **ClassPathXmlApplicationContext:** Loads beans and configuration from the XML file.
- **Bean retrieval:** Beans can be retrieved by name or class.
- **Demonstration:** Outputs all bean names, prints name and age, and runs the game.

4. Important XML Configuration Concepts

Concept	XML Example	Description
Bean Definition	<bean id="name" class="java.lang.String"> <constructor-arg value="Nilesh"/></bean>	Defines a bean and its value
Constructor Wiring	<constructor-arg ref="game" />	Injects another bean via constructor
Component Scan	<context:component-scan base-package="com.example.package" />	Enables annotation-based scanning (optional)
Primitive Types	<bean id="age" class="java.lang.String"><constructor-arg value="24"/></bean>	Defines simple value beans

5. Comparison: XML vs Java Configuration

Aspect	XML Configuration	Java Configuration (Modern)
Syntax	XML files (.xml)	Java classes + annotations
Bean definition	<bean ... />	@Bean methods or @Component
Wiring	<constructor-arg ... />	Constructor injection, @Autowired
Component scanning	<context:component-scan ... />	@ComponentScan
Popularity Today	Rare, mostly legacy	Standard, recommended

6. Why Learn XML Configuration?

- **Legacy projects:** Many older Spring applications use XML for bean definition and wiring.
- **Foundation:** Understanding XML helps you grasp the basics of bean lifecycle and dependency injection in Spring.

- **Interoperability**: You may encounter hybrid projects that use both XML and Java config.

7. Key Takeaways

- **Everything you can do with Java configuration and annotations, you can also do with XML**—but XML is now considered outdated for new projects.
- **Spring's modern approach** is annotation-driven, but XML skills are still valuable for maintenance and migration.
- **XML configuration is verbose and less readable**, but is explicit and was the backbone of early Spring development.

In summary:

- XML configuration allows you to define beans, wire dependencies, and set up your application context in a structured XML file.
- While not common for new Spring projects, it's crucial for understanding legacy code and the foundations of the Spring IoC container.
- Modern Spring development favors Java-based configuration and annotations for their simplicity and readability.

contextConfiguration.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context" xsi:schemaLocation="
        http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd"> <!-- bean
definitions here -->

       <bean id="name" class="java.lang.String">
              <constructor-arg value="Nilesh" />
       </bean>

       <bean id="age" class="java.lang.String">
              <constructor-arg value="24" />
       </bean>

<!--   <context:component-scan base-
package="com.in28minutes.learn_spring_framework.game" />
-->

       <bean id="game" class="com.in28minutes.learn_spring_framework.game.PacmanGame"
/>

       <bean id="gameRunner"
class="com.in28minutes.learn_spring_framework.game.GameRunner">
              <constructor-arg ref="game" />
       </bean>


</beans>
```

XmlConfigurarionContextLauncherApplication.java

```
package com.in28minutes.learn_spring_framework.examples.h1;
```

```
import java.util.Arrays;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.in28minutes.learn_spring_framework.game.GameRunner;



public class XmlConfigurarionContextLauncherApplication {

	public static void main(String[] args) {

		try(var context =
				new
ClassPathXmlApplicationContext("contextConfiguration.xml")) {

			Arrays.stream(context.getBeanDefinitionNames())
				.forEach(System.out::println);

			System.out.println(context.getBean("name"));

			System.out.println(context.getBean("age"));

			context.getBean(GameRunner.class).run();


		}
	}
}
```

PacmanGame.java

```
package com.in28minutes.learn_spring_framework.game;

import org.springframework.stereotype.Component;

@Component
public class PacmanGame implements GamingConsole{
	public void up() {
		System.out.println("up");
	}

	public void down() {
		System.out.println("down");
	}

	public void left() {
		System.out.println("left");
	}

	public void right() {
		System.out.println("right");
	}
}
```

GameRunner.java

```
package com.in28minutes.learn_spring_framework.game;

import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Component;
```

```java
@Component
public class GameRunner {

	private GamingConsole game;

	public GameRunner(@Qualifier("SuperContraGameQualifier") GamingConsole game) {

		this.game = game;
	}

	public void run() {
		System.out.println("Running game: " + game);
		game.up();
		game.down();
		game.left();
		game.right();
	}
}
```

Annotations vs XML Configuration in Spring

1. Ease of Use

- **Annotations**
  - **Very easy to use.**
  - Defined directly on the class, method, or variable—very close to the source code.
  - Syntax is short and concise.
- **XML Configuration**
  - **Cumbersome and verbose.**
  - Requires specifying the full class name (with package) and following strict XML syntax.
  - Setting up things like autowiring can be complex.

2. POJO Cleanliness

- **Annotations**
  - POJOs (Plain Old Java Objects) are **not "clean"**—they include Spring-specific annotations like @Component, @Autowired, etc.
  - This creates a dependency on Spring in your Java code.
- **XML Configuration**
  - **Keeps POJOs clean.**
  - No Spring annotations required in your Java classes; beans are configured entirely via XML.
  - Java code does **not need to change** for Spring configuration.

3. Maintenance

- **Annotations**

    - **Easy to maintain.**
    - Changes can be made directly in the source code next to the relevant class or method.
    - Refactoring (like renaming or moving classes) is simpler—update the annotation and you're done.
- **XML Configuration**
    - **Harder to maintain.**
    - Changing class names, packages, or adding dependencies requires changes in both Java code and XML files.
    - Multiple places to update increases the risk of errors.

4. Usage Frequency

- **Annotations**
    - **Very frequently used** in almost all modern Java/Spring projects.
    - Standard practice for new Spring applications.
- **XML Configuration**
    - **Rarely used** today.
    - Mostly found in older, legacy Spring projects being maintained.

5. Recommendation

- **Either approach is technically fine**, but **be consistent** within your project.
- **Do not mix annotations and XML configuration** in the same project, as it can lead to confusion and maintenance challenges.

6. Debugging Difficulty

- **Annotations**
    - **Harder to debug.**
    - Since configuration is spread across the codebase and not as explicit, a strong understanding of Spring is required to troubleshoot issues.
- **XML Configuration**
    - **Medium difficulty.**
    - Configuration is verbose and explicit, making it easier to trace and debug problems.

7. Summary Table (from Slides)

Aspect	Annotations	XML Configuration
Ease of use	Very easy, close to source	Cumbersome
Conciseness	Short and concise	No
Clean POJOs	No (polluted with annotations)	Yes (no Spring code in POJOs)
Easy to Maintain	Yes	No
Usage Frequency	Almost all recent projects	Rarely
Recommendation	Be consistent, use one approach	Be consistent, use one approach
Debugging	Hard	Medium

8. Key Takeaways

- **Annotations/Java config is preferred** for new projects due to ease of use and maintainability.
- **XML config keeps POJOs framework-agnostic** but is harder to maintain and less popular now.
- **Choose one approach for your project and stick to it** for clarity and maintainability.
- **XML configuration may be easier to debug** for beginners, as everything is explicitly spelled out.

In summary:
Annotations provide a concise, modern, and easy-to-maintain way to configure Spring, but "pollute" your Java code with framework dependencies. XML keeps your Java code clean but is verbose, cumbersome, and mostly used in legacy projects. Whichever you choose, be consistent throughout your project.

Spring Stereotype Annotations: @Component, @Service, @Repository, @Controller

1. What Are Stereotype Annotations?

- **Stereotype annotations** are special annotations in Spring used to declare beans, define their roles, and organize your codebase semantically.
- The base annotation is @Component. All others are specializations of @Component.

2. The Main Stereotype Annotations

Annotation	Typical Usage	Description
@Component	Generic Spring bean	Used for any class you want managed by Spring.
@Service	Business logic/service layer	Indicates the class contains business logic.
@Repository	Data access layer	Indicates the class interacts with a database.
@Controller	Web layer/controller	Used for web controllers (MVC or REST APIs).

3. Example from the Code

- **BusinessCalculationService**
  - Annotated with @Service: contains business logic.
- **MongoDbDataService** and **MySQLDataService**

    - Annotated with @Repository: interact with data sources (databases).
- **No @Controller in this example**, but you would use it for web controllers.

4. Why Use the Most Specific Annotation?

- **Clarity:**
  Makes your codebase easier to understand—other developers (and you, later) can instantly see the intent of each class.
- **Framework Features:**
  Spring can provide *additional behavior* for certain stereotypes:
    - For @Repository: automatic JDBC exception translation.
    - For @Service and @Controller: can be used by AOP (Aspect-Oriented Programming) for additional cross-cutting concerns like transactions, security, etc.
- **AOP Support:**
  You can target specific stereotypes in your aspects (e.g., logging only for service methods).
- **Best Practice:**
  Always use the **most specific annotation possible** to give Spring (and your team) more information.

5. When to Use Which Annotation?

Use Case	Annotation
Business logic/service layer	@Service
Database/repository layer	@Repository
Web controller	@Controller
None of the above	@Component

6. Practical Benefits

- **@Repository**: Enables features like exception translation.
- **@Controller**: Enables web request mapping, response handling, etc.
- **@Service**: Semantically separates business logic from other beans.
- **@Component**: For generic Spring beans that don't fit the above categories.

7. Summary Table (from Slides)

Annotation	Use for...
@Component	Any generic Spring bean
@Service	Business logic
@Controller	Web controllers (web apps, REST APIs)
@Repository	Data access / repository classes

8. Key Takeaways

- **@Component** is the base stereotype—can be used for any bean.
- **@Service, @Repository, @Controller** are specializations for specific layers of your application.

- **Use the most specific stereotype annotation possible** to improve clarity and leverage Spring features.
- These annotations all make the class a Spring bean, enabling dependency injection and lifecycle management.

In summary:
Use @Service for business logic, @Repository for data access, @Controller for web controllers, and @Component for generic beans. Favor the most specific annotation to help both Spring and your team understand the intent and enable extra framework features.

RealWorldSpringContextLauncherApplication.java

```
package com.in28minutes.learn_spring_framework.examples.c1;

import java.util.Arrays;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan
public class RealWorldSpringContextLauncherApplication {

	public static void main(String[] args) {

		try(var context =
				new AnnotationConfigApplicationContext
				   (RealWorldSpringContextLauncherApplication.class)) {

			Arrays.stream(context.getBeanDefinitionNames())
				.forEach(System.out::println);

			System.out.print(

	context.getBean(BusinessCalculationService.class).findMax());
		}
	}
}
```

BusinessCalculationService.java

```
package com.in28minutes.learn_spring_framework.examples.c1;

import java.util.Arrays;

// import org.springframework.stereotype.Component;
import org.springframework.stereotype.Service;

// @Component
@Service
public class BusinessCalculationService {

	private DataService dataService;

	public BusinessCalculationService(DataService dataService) {
		super();
		this.dataService = dataService;
	}
```

```
	public int findMax() {
		return Arrays.stream(dataService.retrieveData()).max().orElse(0);
	}
}
```

MongoDbDataService.java

```
package com.in28minutes.learn_spring_framework.examples.c1;


import org.springframework.context.annotation.Primary;
// import org.springframework.stereotype.Component;
import org.springframework.stereotype.Repository;

// @Component
@Repository
@Primary
public class MongoDbDataService implements DataService{

	@Override
	public int[] retrieveData() {
		return new int[] { 11, 22, 33, 44, 55 };
	}

}
```

MySQLDataService.java

```
package com.in28minutes.learn_spring_framework.examples.c1;


// import org.springframework.stereotype.Component;
import org.springframework.stereotype.Repository;

// @Component
@Repository
public class MySQLDataService implements DataService{

	@Override
	public int[] retrieveData() {
		return new int[] { 1, 2, 3, 4, 5 };
	}

}
```

Quick Review of Important Spring Annotations

1. Configuration and Bean Definition

Annotation	Description
@Configuration	Indicates a class declares one or more @Bean methods. Used for Java-based configuration. The Spring container processes these classes to generate bean definitions.
@Bean	Indicates a method produces a bean to be managed by the Spring container. Defined inside a @Configuration class.
@ComponentScan	Defines which packages to scan for Spring components (@Component, etc.). If no package is specified, scanning starts from the package of the class declaring this annotation (including subpackages).

2. Stereotype Annotations

Annotation	Description
@Component	Generic annotation indicating a class is a Spring component (bean). Base for all stereotype annotations.
@Service	Specialization of @Component. Indicates the class contains business logic (Service layer).
@Controller	Specialization of @Component. Indicates the class is a controller (web or REST API layer).
@Repository	Specialization of @Component. Indicates the class handles data access or database operations.

3. Bean Selection and Wiring

Annotation	Description
@Primary	Gives a bean preference when multiple candidates are available for autowiring a single-valued dependency.
@Qualifier	Used on fields or parameters to specify exactly which bean to inject, by name or qualifier.

4. Bean Initialization and Scope

Annotation	Description
@Lazy	Specifies that a bean should be lazily initialized (created only when first requested). Without it, beans are eagerly initialized at context startup.
@Scope	Sets the scope of a bean. Common values: ConfigurableBeanFactory.SCOPE_PROTOTYPE (new instance each time) and singleton (default, one per context).

5. Bean Lifecycle Callbacks

Annotation	Description
@PostConstruct	Marks a method to run after dependency injection, for initialization logic.
@PreDestroy	Marks a method to run just before the bean is destroyed, for cleanup and releasing resources.

6. Jakarta CDI (Contexts and Dependency Injection) Annotations

Annotation	Description
@Named	CDI annotation (Jakarta EE), similar in role to @Component (declares a bean by name).
@Inject	CDI annotation, equivalent to Spring's @Autowired (performs dependency injection).

7. Practical Tips and Takeaways

- Use @Configuration and @Bean for Java-based Spring configuration.
- Use @Component, @Service, @Controller, and @Repository to clearly define the roles of your beans.
- Use @Primary and @Qualifier for precise control over which beans are injected.
- Use @Lazy and @Scope to manage when and how many bean instances are created.
- Use @PostConstruct and @PreDestroy to handle initialization and cleanup in bean lifecycles.
- Spring supports Jakarta CDI annotations (@Named, @Inject) for compatibility and standards-based development.

In summary:
Spring provides a wide range of annotations to simplify configuration, clarify bean roles, control bean scope and lifecycle, and support standards-based development via CDI annotations. Understanding these annotations is essential for writing clear, maintainable, and robust Spring applications.

Quick Review: Key Spring Concepts

1. Dependency Injection (DI)

- **Definition:**
  The process by which Spring identifies beans, their dependencies, and wires them together automatically.
- **Also known as:**
  Inversion of Control (IoC)—because the responsibility for creating and connecting objects shifts from your code to the Spring Framework.

2. Types of Dependency Injection

Type	How it works
Constructor Injection	Dependencies are provided via the class constructor. Spring creates the bean by calling its constructor and passing in dependencies.
Setter Injection	Dependencies are set using setter methods. Spring calls the bean's setter methods after instantiating it.
Field Injection	Dependencies are injected directly into fields (using reflection), with no constructor or setter needed.

3. Spring IoC Container

- **What is it?**
  The core part of Spring that manages the lifecycle, configuration, and assembly of beans.
- **Main types:**
  - **BeanFactory:**
    The most basic IoC container. Rarely used directly.
  - **ApplicationContext:**
    The standard, feature-rich container used in almost all Spring applications.

- Adds enterprise features (AOP, internationalization, etc.)
- Preferred for web and enterprise apps.

4. Spring Beans

- **Definition:**
  Any object that is managed by the Spring IoC container.
- **Lifecycle:**
  Created, configured, and destroyed by Spring according to configuration.

5. Auto-wiring

- **Definition:**
  The process by which Spring automatically finds and injects the correct dependencies into beans.
- **Benefit:**
  Reduces boilerplate and manual wiring, making code easier to maintain.

6. Summary Table (from Slides)

Concept	Description
Dependency Injection	Identify beans, their dependencies, and wire them together (provides IoC)
Constructor Injection	Dependencies set via bean constructor
Setter Injection	Dependencies set via setter methods
Field Injection	Dependencies injected directly into fields (reflection)
IoC Container	Manages Spring beans and their lifecycle
BeanFactory	Basic IoC container
ApplicationContext	Advanced IoC container, enterprise features, preferred in modern Spring
Spring Beans	Objects managed by Spring
Auto-wiring	Spring automatically wires dependencies for a bean

7. Key Takeaways

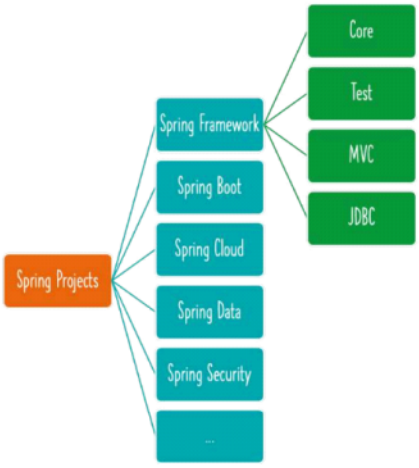
- **Spring's core:** Dependency injection and IoC container.
- **Constructor, setter, and field injection:** Multiple ways to provide dependencies.
- **ApplicationContext** is the standard container for modern Spring apps.
- **Auto-wiring** simplifies dependency management.
- **Spring beans** are just regular objects, but managed by Spring.

In summary:
Spring manages your objects (beans), injects dependencies automatically (DI/IoC), and provides robust containers (ApplicationContext) and auto-wiring to simplify and organize your application structure.

Spring Big Picture: Framework, Modules, and Projects

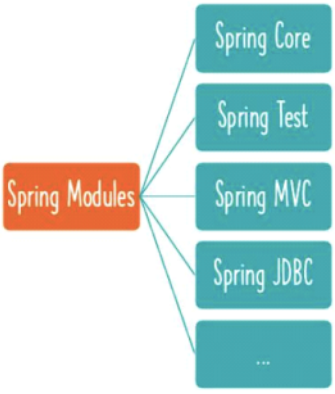
Spring Big Picture - Framework, Modules and Projects

- **Spring Core** : IOC Container, Dependency Injection, Auto Wiring, ..
  - These are the fundamental building blocks to:
    - Building web applications
    - Creating REST API
    - Implementing authentication and authorization
    - Talking to a database
    - Integrating with other systems
    - Writing great unit tests
- Let's now get a Spring Big Picture:
  - **Spring Framework**
  - **Spring Modules**
  - **Spring Projects**



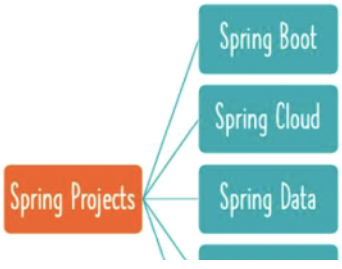
Spring Big Picture - Framework and Modules

- Spring Framework contains multiple **Spring Modules**:
  - **Fundamental Features**: Core (IOC Container, Dependency Injection, Auto Wiring, ..)
  - **Web**: Spring MVC etc (Web applications, REST API)
  - **Web Reactive**: Spring WebFlux etc
  - **Data Access**: JDBC, JPA etc
  - **Integration**: JMS etc
  - **Testing**: Mock Objects, Spring MVC Test etc
- **No Dumb Question**: Why is Spring Framework divided into Modules?
  - Each application can choose modules they want to make use of
  - They do not need to make use of everything in Spring framework!



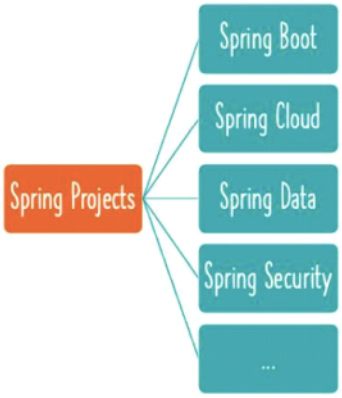
Spring Big Picture - Spring Projects

- Application architectures evolve continuously
  - Web > REST API > Microservices > Cloud > ...
- Spring evolves through **Spring Projects**:
  - **First Project**: Spring Framework
  - **Spring Security**: Secure your web application or REST API or microservice



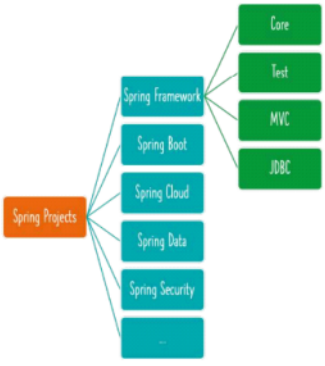
Spring Big Picture - Spring Projects

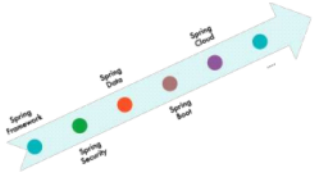
- Application architectures evolve continuously
  - Web > REST API > Microservices > Cloud > ...
- Spring evolves through **Spring Projects**:
  - **First Project**: Spring Framework
  - **Spring Security**: Secure your web application or REST API or microservice
  - **Spring Data**: Integrate the same way with different types of databases : NoSQL and Relational
  - **Spring Integration**: Address challenges with integration with other applications
  - **Spring Boot**: Popular framework to build microservices
  - **Spring Cloud**: Build cloud native applications



Spring Big Picture - Framework, Modules and Projects

- **Hierarchy**: Spring Projects > Spring Framework > Spring Modules
- Why is Spring Eco system popular?
  - **Loose Coupling**: Spring manages creation and wiring of beans and dependencies
    - Makes it easy to build loosely coupled applications
    - Make writing unit tests easy! (Spring Unit Testing)
  - **Reduced Boilerplate Code**: Focus on Business Logic
    - Example: No need for exception handling in each method!
      - All Checked Exceptions are converted to Runtime or Unchecked Exceptions
  - **Architectural Flexibility**: Spring Modules and Projects
    - You can pick and choose which ones to use (You DON'T need to use all of them!)
  - **Evolution with Time**: Microservices and Cloud
    - Spring Boot, Spring Cloud etc!





1. Spring Core Concepts

- **Spring Core** is the foundation of the Spring ecosystem.
  - Provides:
    - **IOC Container** (Inversion of Control)
    - **Dependency Injection**
    - **Auto Wiring**
- **Why these matter:**
  - They are the fundamental building blocks for:
    - Building web applications
    - Creating REST APIs
    - Implementing authentication/authorization

        - Talking to databases
        - Integrating with other systems
        - Writing unit tests

2. Spring Framework Structure

A. Spring Framework

- The main platform, containing multiple **modules**.
- **Modules** let you pick only what your application needs.

B. Spring Modules

- **Core**: IOC Container, Dependency Injection, Autowiring (fundamentals)
- **Web**:
    - **Spring MVC** – For web applications and REST APIs
- **Web Reactive**:
    - **Spring WebFlux** – For building reactive applications
- **Data Access**:
    - **Spring JDBC**, **Spring JPA** – For database interactions
- **Integration**:
    - **Spring JMS** – For integrating with messaging systems
- **Testing**:
    - **Spring Test** – For unit/integration testing, mock objects, etc.

Why modules?

- **Flexibility**:
    - You only use what you need.
    - Applications don't need to use everything in Spring.
    - Example: For a web app talking to a database, you might use only Spring MVC, JDBC, and Core modules.

3. Spring Projects

- **Spring's ecosystem** has evolved to meet modern needs through **Spring Projects**.
    - **Spring Framework**: The foundational project.
    - **Spring Security**: For authentication and authorization (web, REST, microservices).
    - **Spring Data**: Uniform way to access relational and NoSQL databases.
    - **Spring Integration**: Helps integrate your app with other systems.
    - **Spring Boot**: Streamlines building microservices (opinionated defaults, quick setup).
    - **Spring Cloud**: Tools for building cloud-native, distributed systems.
- **Architectural Evolution**:
    - The Spring ecosystem adapts as software architecture evolves (web apps → REST → microservices → cloud).

4. Hierarchy of the Spring Ecosystem

- **Spring Projects**
    - **Spring Framework**
        - **Spring Modules**

This hierarchy shows how the ecosystem is structured and how projects build on top of the framework and its modules.

5. Why is the Spring Ecosystem so Popular?

Reason	Explanation
Loose Coupling	Spring manages object creation and wiring, making your app modular, maintainable, and testable.
Easier Unit Testing	Dependency injection and loose coupling make it simple to write tests.
Reduced Boilerplate	You focus on business logic. Spring handles repetitive infrastructure code (e.g., exception handling).
Architectural Flexibility	You pick only the modules and projects you need.
Continuous Evolution	Spring adapts to new architectures (microservices, cloud, etc.) through new projects like Spring Boot and Spring Cloud.

Example:

- Earlier, database code required 50-60 lines per query. With Spring JDBC/JPA, it can be done in ~5 lines.

6. Key Takeaways

- **Spring is modular and flexible:** Use only what your app needs.
- **Spring adapts to modern needs:** Keeps up with REST, microservices, cloud, etc.
- **Spring remains popular** because it evolves, reduces code, and makes apps easier to maintain and test.

In summary:
Spring started as a core framework with dependency injection and evolved into a rich ecosystem of modules and projects. Its modularity, flexibility, and continual evolution make it the number one choice for modern Java application development, from web apps to microservices and cloud-native systems.

Learning Pledge: Making Learning a Habit

1. Why a Learning Pledge?

- **Technology constantly evolves:**
  - New frameworks, tools, and processes emerge all the time.
  - What exists now may not have existed 10 years ago.
- **Lifelong learning is essential:**
  - To stay ahead in technology, you must continuously learn.

2. Lessons from Experience

- The instructor shares 25 years of experience in tech:
  - Has seen many languages, frameworks, tools, and evolving software processes.
  - The key to staying relevant is **consistent learning**.

3. Building the Habit of Learning

- **Consistency matters:**
  - Building any good habit requires doing it regularly over time.
  - Common advice: do something for 30 days to build a habit.
  - **in28Minutes motto:** Learn for 28 minutes a day for 28 days.

4. The in28Minutes Learning Pledge

- **What is it?**
  - A personal commitment to learn for 28 minutes every day for 28 days.
- **Why?**
  - To develop the habit of daily learning.
  - To keep up with evolving technology.
- **How to join?**
  - Start today.
  - Share your pledge in the Udemy Q&A if you wish.

5. Key Takeaways

- Technology will keep changing; you need to change and learn with it.
- Make learning a daily habit to stay ahead.
- The **Learning Pledge** is a tool to help you build that habit and achieve long-term success in your technology career.

In summary:
To stay successful and relevant in tech, commit to learning for 28 minutes every day for 28 days. Make learning a habit, and evolve as technology evolves!

Learning Maven with Spring and Spring Boot

18 July 2025 15:53

Introduction to Maven: The Java Build Tool

1. Why Do You Need a Build Tool?

- To become a great programmer, you must know how to build your projects efficiently.
- In the Java ecosystem, the two most popular build tools are:
  - **Maven**
  - **Gradle**
- This section focuses on **Maven**.

2. What is Maven?

- Officially, **Apache Maven** is a software project management and comprehension tool based on the concept of a Project Object Model (POM).
- Maven can manage:
  - A project's build process
  - Reporting
  - Documentation
  - All from a central piece of information (the POM file)
- The official definition is generic—so it's better to understand Maven by looking at what developers use it for.

3. What Does Maven Help With?

- **Creating new projects**: Quick project scaffolding.
- **Managing dependencies**:
  - Add, update, or remove libraries (like Spring Boot, Spring MVC, Hibernate, etc.)
  - Manage versions of all dependencies.
- **Building artifacts**:
  - Build **JAR** files (Java ARchives) or **WAR** files (Web ARchives) for deployment.
  - Build Docker images (with plugins and advanced configurations).
- **Running and deploying applications**:
  - Run your app locally (e.g., on Tomcat, Jetty, or other servers).
  - Deploy to various environments: test, QA, production, etc.
- **Running unit tests**:
  - Easily run and check all your unit tests as part of the build process.

4. Why is Maven Important?

- It simplifies and automates repetitive tasks in Java project development.
- Helps manage complex dependency trees and version conflicts.
- Ensures consistent builds and deployments across different environments.

5. What Will You Learn?

- How Maven can help with:
  - Creating and managing projects
  - Handling dependencies
  - Building and deploying applications
  - Running tests
- You will see practical demonstrations of these features in this section.

6. Key Takeaways

- **Maven is an essential tool** for modern Java development.
- It helps manage the complete lifecycle of your project: from creation, through building and testing, to deployment.
- Learning Maven will make you a more effective and professional Java developer.

In summary:
Maven is a powerful, essential build and project management tool in the Java ecosystem. It helps you create, build, test, manage dependencies, and deploy your applications with ease and consistency.

Getting Started: Creating a Simple Maven Project

1. Spring Initializr – start.spring.io

- Use start.spring.io to quickly generate a new Maven project.
- **Select "Maven Project"** as the project type.
- **Language:** Java.
- **Spring Boot Version:**
  - Choose the latest released version (avoid "SNAPSHOT" versions, as they are under development).
  - Example: If 3.0.0 or newer is available, use that.

2. Key Maven Project Settings

- **Group ID:**
  - Acts like a package name for your project (e.g., com.in28minutes).

    - Used for namespacing and identifying your project uniquely.
- **Artifact ID:**
    - The name of your application or module (e.g., learn-maven).
    - Other projects use this ID to reference your project as a dependency.
- Leave other fields as default unless you have specific requirements.

3. Generating and Importing the Project

- Click "Generate" to download the ZIP file.
- Extract the ZIP file to a folder.
- **Import into Eclipse:**
    - Go to File → Import → Maven → Existing Maven Projects.
    - Select the extracted project folder.
    - Finish the import process.
- **Wait for dependencies to download** (the first time may take a few minutes).

4. Typical Maven Project Structure

- src/main/java: Your main Java source code.
- src/main/resources: Resources for your application (e.g., properties files).
- src/test/java: Unit and integration test code.
- src/test/resources: Resources needed for tests.

5. pom.xml – The Heart of Maven

- **pom.xml** stands for "Project Object Model."
- It is a central XML file that:
    - Defines your project's metadata (group ID, artifact ID, version).
    - Lists all dependencies and plugins.
    - Controls how your project is built and managed.
- **Understanding pom.xml is key to mastering Maven.**

6. What's Next?

- You have now created and imported a Maven project into your IDE.
- The next step is to dive deeper into the pom.xml file to understand its structure and how it controls your Maven project.

In summary:
You used Spring Initializr to generate a Maven project, set the group and artifact IDs, imported it into Eclipse, and learned about the standard Maven project structure. The pom.xml file is central to Maven—mastering it is crucial for effective Java project management.

Understanding pom.xml (Project Object Model) and Maven Dependencies

1. What is pom.xml?

- The **Project Object Model** (pom.xml) is the heart of any Maven project.
- It is an XML file that:
  - Defines project metadata (groupId, artifactId, version, name, description, etc.)
  - Lists all dependencies (libraries and frameworks your project needs)
  - Configures plugins, build instructions, and more.

2. Maven Dependencies

- **Dependencies** are external libraries/frameworks your project needs to compile, run, and test.
- **Where are they defined?**
  - Inside the <dependencies> section of your pom.xml.
- **Example from your pom.xml:**

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

- **What happens when you add a dependency?**
  - Maven automatically downloads not just the specified library, but also all of its dependencies (called **transitive dependencies**).
  - For example, spring-boot-starter-web pulls in Spring MVC, Tomcat, Jackson, etc., automatically.

3. Transitive Dependencies

- When you add a dependency (e.g., spring-boot-starter-web), Maven checks what other libraries it needs, and downloads those as well.
- This means you only need to specify the main dependencies—Maven handles the rest.
- Example: Adding spring-boot-starter-test brings in JUnit, Mockito, Hamcrest, and more for testing.

4. Typical Structure of pom.xml

Element	Purpose	Example/Notes
<groupId>	Like a Java package name. Uniquely identifies your org/project	com.in28minutes
<artifactId>	Name of your project/module	learn-maven
<version>	Project version	0.0.1-SNAPSHOT

<dependencies>	List of required libraries	See above
<build> <plugins>	Plugins for building, packaging, running, etc.	spring-boot-maven-plugin for Spring Boot apps
<properties>	Project-wide properties (e.g., Java version)	<java.version>17</java.version>
<parent>	Inherits config from a parent POM (like Spring Boot's starter)	spring-boot-starter-parent

5. Dependency Hierarchy & Management

- You can view all the dependencies and how they are brought into your project (direct vs. transitive) in your IDE (like Eclipse or IntelliJ).
- Example:
  - spring-boot-starter-web brings in:
    - spring-webmvc, spring-boot-starter-json, spring-boot-starter-tomcat, etc.
  - spring-boot-starter-test brings in:
    - junit-jupiter, mockito-core, etc.

6. Difference: Maven Dependencies vs. Spring Dependencies

- **Maven dependencies:**
  - Defined in pom.xml, refer to libraries/frameworks at the project level.
  - Example: Adding Spring Boot, Hibernate, etc.
- **Spring dependencies:**
  - Refer to beans and components within your Java code (e.g., when using @Autowired or @Inject).
  - Managed at runtime by the Spring Framework.

7. Key Takeaways

- **pom.xml** is essential for managing your project's dependencies, build process, and metadata.
- Adding a dependency in pom.xml brings in the library and all its transitive dependencies.
- Maven handles downloading and including all required JARs, so you don't need to manage them manually.
- Understanding pom.xml is crucial for working with Maven projects.

In summary:
The pom.xml file manages all your project's dependencies and build configuration. By adding a few key dependencies, Maven takes care of downloading everything your project needs (including transitive dependencies), making your development process much easier and more reliable.

Understanding the Parent POM in Maven (Spring Boot Example)

1. What is a Parent POM?

- In Maven, a **parent POM** allows you to inherit configuration, properties, dependency versions, and plugin settings from another project.
- In Spring Boot projects, the parent POM is typically spring-boot-starter-parent.

2. What Do You Get from the Spring Boot Parent POM?

- **Centralized Configuration:**
  The parent POM brings in hundreds or thousands of lines of configuration so you don't have to specify everything in your own pom.xml.
- **Inheritance Chain:**
  - spring-boot-starter-parent itself inherits from spring-boot-dependencies, which contains most of the dependency management settings.

3. Exploring the Effective POM

- **Effective POM:**
  This is the full, merged view of your own pom.xml plus everything you inherit from your parent POMs.
- **How to view it:**
  - Use the Maven command: mvn help:effective-pom
  - In Eclipse, open the **Effective POM** tab.
- **Size:**
  - Your pom.xml may be ~70 lines, but the effective POM can be thousands of lines (e.g., 6,935 lines).
  - All inherited settings, dependencies, and plugins are included.

4. Dependency Management

- **No Need to Specify Versions:**
  - Thanks to the parent POM, you typically **do not specify versions** for commonly used dependencies.
  - Example:
    ```
    <dependency>
      <groupId>org.hibernate.orm</groupId>
      <artifactId>hibernate-core</artifactId>
      <!-- No version specified -->
    </dependency>
    ```
  - Maven uses the version defined in the parent POM's <dependencyManagement> section (e.g., Hibernate 6.1.4.Final).
- **How it works:**
  - The <dependencyManagement> section in the parent POM lists versions.
  - The <dependencies> section in your own POM lists what you want to use.
  - Maven combines them: it uses your list, but fills in versions from the parent.

5. Plugins and Properties

- **Plugins:**

The parent POM also provides default Maven plugins and their configurations (for building, testing, packaging, etc.).
- **Properties:**
  Important settings like java.version (e.g., 17) are also inherited from the parent POM.

6. Practical Benefits

- **Less Boilerplate:**
  You write less configuration in your own pom.xml.
- **Consistency:**
  All projects using the same parent POM have consistent dependency versions and plugin settings.
- **Easy Upgrades:**
  Change parent POM version to upgrade dependencies across your project.

7. Key Takeaways

- **Spring Boot parent POM** gives you:
  - **Dependency management** (no need to specify versions)
  - **Shared properties** (like Java version)
  - **Pre-configured plugins** (for build, test, run)
- **Effective POM** shows everything you inherit and use.
- **Dependency versions** are resolved from the parent, making your pom.xml much simpler and easier to maintain.

In summary:
The Spring Boot parent POM streamlines your project by handling dependency versions, plugin configurations, and key properties for you. By inheriting from it, you can focus on what's unique to your project and let Maven and Spring Boot manage the rest.

Understanding Maven Dependency Tree and POM Metadata

1. Maven Dependency Tree: What Is It and Why Use It?

- The mvn dependency:tree command shows a **hierarchical view** of all the dependencies in your project, including both:
  - **Direct dependencies** (what you declare in your <dependencies>)
  - **Transitive dependencies** (what your dependencies depend on)
- **Why is this useful?**
  - To see **exactly what libraries** are included in your build
  - To debug **version conflicts** or unexpected dependencies
  - To verify that test-only dependencies are not leaking into your main code

2. Example Output: What Does the

Dependency Tree Show?

Sample Output Breakdown

```
[INFO] com.in28minutes:learn-maven:jar:0.0.1-SNAPSHOT
[INFO] +- org.springframework.boot:spring-boot-starter:jar:3.5.3:compile
[INFO] |  +- org.springframework.boot:spring-boot:jar:3.5.3:compile
[INFO] |  |  \- org.springframework:spring-context:jar:6.2.8:compile
...
[INFO] \- org.springframework.boot:spring-boot-starter-test:jar:3.5.3:test
[INFO]    +- org.junit.jupiter:junit-jupiter:jar:5.12.2:test
[INFO]    +- org.mockito:mockito-core:jar:5.17.0:test
...
[INFO] BUILD SUCCESS
```

- **Root node:** Your project (com.in28minutes:learn-maven:jar:0.0.1-SNAPSHOT)
- **Direct dependencies:**
  - spring-boot-starter
  - spring-boot-starter-web
  - spring-boot-starter-test (with <scope>test</scope>)
- **Transitive dependencies:**
  - Each starter brings in many more dependencies.
    - Example: spring-boot-starter-web brings in Jackson, Tomcat, Spring MVC, etc.
    - spring-boot-starter-test brings in JUnit, Mockito, AssertJ, etc.

3. Understanding POM Metadata

Element	Purpose and Importance
<groupId>	Like a Java package name; uniquely identifies your organization or project group
<artifactId>	Name of your project/module; used by other projects to depend on your artifact
<version>	The version of your project; important for publishing and versioning artifacts
<name>	Human-readable name for your project
<description>	Short description of your project

- **Why are these important?**
  - When you publish your project (e.g., to a Maven repository), other projects use your groupId, artifactId, and version to add your project as a dependency.
  - These fields are also used in the dependencies you add from other libraries.

4. How Maven Resolves Dependencies

- When you specify a dependency in <dependencies>, Maven:
  1. Looks for it in the Maven Central Repository (or other configured repositories)
  2. Downloads the specified artifact and all its transitive dependencies
  3. Adds them to your project's classpath according to their scope (e.g., compile, test)
- **Scopes:**
  - compile: Used in main code

    - test: Only available in test code (e.g., JUnit, Mockito)
    - Others: provided, runtime, etc.

5. Investigating and Troubleshooting

- **Dependency Tree:**
    - Use mvn dependency:tree to visualize and debug what is actually included in your build.
- **Effective POM:**
    - Use mvn help:effective-pom to see the final, merged configuration after inheriting from parent POMs.

6. Key Takeaways

- **The POM file** is central to how Maven manages your project's dependencies, metadata, and build process.
- **groupId, artifactId, version**: These uniquely identify both your project and all dependencies you use.
- **dependency:tree**: Essential tool for seeing all dependencies (direct and transitive) and for troubleshooting version conflicts or unexpected inclusions.
- **Scopes** help control where dependencies are available (main code vs. test code).
- **Transitive dependencies** mean you often get much more than you explicitly declare.

In summary:
Maven's dependency management, powered by the POM's metadata and dependency declarations, makes it easy to add powerful libraries to your project and to see exactly what is included via the dependency:tree command. Understanding this system is essential for professional Java development with Maven.

Maven Build Lifecycle: What Happens When You Run mvn install?

1. What is the Maven Build Lifecycle?

- The **Maven Build Lifecycle** is a **sequence of standard build phases** that are executed in order when you run Maven commands.
- Each phase performs a specific task in the build process.

2. Main Maven Build Phases

Phase	What it does
validate	Checks if the project structure and configuration are correct.
compile	Compiles the main Java source code (src/main/java).
test	Compiles and runs tests (src/test/java). Uses frameworks like JUnit.
package	Packages the compiled code into a distributable format (e.g., JAR).
verify	Runs any checks to verify the package is valid and meets quality criteria.
install	Copies the built JAR/WAR to your **local Maven repository**.
deploy	(If configured) Publishes the artifact to a remote repository/server.

- **Note:**
  There are also additional steps (like integration-test) in more advanced builds.

3. Typical Steps in mvn install Output

- **Copy resources:**
  Resources from src/main/resources are copied to the target/classes directory.
- **Compile main code:**
  Java files in src/main/java are compiled to target/classes.
- **Compile test code:**
  Java files in src/test/java are compiled to target/test-classes.
- **Run tests:**
  All unit tests are executed (using Surefire plugin and frameworks like JUnit).
- **Package:**
  A JAR file (or WAR, if web app) is created in the target folder.
- **Repackage (Spring Boot):**
  The spring-boot-maven-plugin may repackage the JAR to make it executable.
- **Install:**
  The final artifact (JAR/WAR) is copied to your local Maven repository (usually under ~/.m2/repository).
- **Test reports and artifacts:**
  Test results and reports are stored in the target folder.

4. The target Folder

- After running mvn install, the target folder contains:
  - classes/ — compiled main Java classes
  - test-classes/ — compiled test classes
  - learn-maven-0.0.1-SNAPSHOT.jar — the packaged JAR file
  - surefire-reports/ — unit test execution reports
  - .original JAR file (if using Spring Boot repackage)
- **You can explore these files** to see exactly what Maven has built.

5. Why is the Lifecycle Important?

- **Consistency:**
  Every Maven build follows the same sequence, ensuring repeatable, reliable builds.
- **Automation:**
  One command (mvn install) handles compiling, testing, packaging, and installing your app.

- **Extensibility:**
  You can add plugins to hook into any phase (e.g., code analysis, documentation generation).

6. Key Takeaways

- **Maven's lifecycle** automates all steps from validation to packaging and installation.
- **Running** `mvn install` executes all necessary steps to build, test, package, and locally install your application.
- **Artifacts and reports** are placed in the `target` folder for inspection.
- This process is **standardized** for all Maven projects, making it easy to work on different Java projects.

In summary:
The Maven Build Lifecycle is a series of standardized phases that automate compiling, testing, packaging, and installing your application. Running `mvn install` ensures your app is built, tested, and the final artifact is available in your local Maven repository for use by other projects.

How Does Maven Work? (Repositories and Dependency Resolution)

1. Convention over Configuration

- **Maven uses a standard folder structure:**
  - `src/main/java` for main Java code
  - `src/main/resources` for main resources
  - `src/test/java` for test Java code
  - `src/test/resources` for test resources
- **Benefit:**
  - Consistency across Java projects
  - Easy to navigate new projects—everyone knows where to find code, resources, and tests

2. Maven Central Repository

- **What is it?**
  - The main online storage (repository) for Java libraries (JAR files)
  - Libraries are indexed by `groupId` and `artifactId`
- **How dependencies work:**

- When you declare dependencies in your pom.xml, Maven downloads the required JARs from Maven Central
- Example: repo1.maven.org/maven2

- **You can browse Maven Central** to see available versions and artifacts for any library

3. How Maven Resolves Dependencies

- **Default behavior:**
  - Maven first looks in the **local Maven repository** on your computer (usually ~/.m2/repository)
  - If not found locally, Maven downloads from **remote repositories** (like Maven Central)
- **Process:**
  1. You add a dependency in pom.xml
  2. Maven checks if it exists in your local repo
  3. If not, Maven downloads it (and its dependencies) from the remote repo
  4. Downloaded JARs are cached in your local repo for future builds

4. Additional Repositories

- **Not all artifacts are on Maven Central.**
  - Release candidates (RC), milestones (M), and snapshots may be hosted elsewhere
- **Custom repositories:**
  - You can add extra repositories in pom.xml under <repositories> and <pluginRepositories>
  - Example: **Spring Milestones Repository** for pre-release Spring libraries
- **Maven will look in these repositories if a dependency isn't found in Central**

5. Plugins and Plugin Repositories

- **Plugins** (e.g., spring-boot-maven-plugin) are also downloaded from repositories
- You can configure plugin repositories in pom.xml for non-standard or milestone plugin versions

6. Local Maven Repository

- **Location:**
  - Usually at ~/.m2/repository (Linux/macOS) or C:\Users\<user>\.m2\repository (Windows)
- **Purpose:**
  - Caches all downloaded JARs and plugins for faster future builds
  - Maven projects use these local files instead of downloading every time

7. Summary Flow

1. **You add a dependency** in pom.xml
2. **Maven checks local repository** for the JAR
3. If not found, **Maven checks remote repositories** (Central and any custom ones)
4. **JARs are downloaded and cached locally**
5. **Your project uses the local copies** for builds

Key Takeaways

- Maven provides **standardization** (convention over configuration) for project structure
- **Dependencies** are resolved automatically from local or remote repositories
- **Central Maven Repository** is the default, but you can add others for special versions
- **Local repository** makes builds faster and offline-friendly after first download

In summary:
Maven automates dependency management by downloading libraries from central (and optionally custom) repositories, caching them locally, and using a consistent project structure. This makes Java project setup, maintenance, and collaboration much simpler and more reliable.

Essential Maven Commands: What They Do and When to Use Them

1. Check Your Maven and Java Setup

- **Command:** mvn --version
- **Purpose:** Shows Maven version, Maven home directory, Java version, OS details.
- **Use it when:** You want to confirm your Maven/Java installation and environment.

2. Compile Source Code

- **Command:** mvn compile
- **Purpose:** Compiles your main Java source files (src/main/java).
- **What it does:**
  - Checks if sources have changed.
  - Only recompiles if necessary.
- **Output:** Compiled .class files in target/classes.

3. Compile Test Code

- **Command:** mvn test-compile
- **Purpose:** Compiles both main and test Java source files (src/main/java and src/test/java).
- **Output:**
  - Main classes in target/classes.
  - Test classes in target/test-classes.

4. Clean Build Artifacts

- **Command:** mvn clean
- **Purpose:** Deletes the target directory (removes all compiled classes, test reports, and built JARs).
- **When to use:** Before a fresh build or to resolve strange build issues.

5. Run Unit Tests

- **Command:** mvn test
- **Purpose:**
  - Compiles source and test files (if needed).
  - Runs all unit tests (e.g., with JUnit).
- **Output:**
  - Test results in the console and in target/surefire-reports.

6. Package Your Application

- **Command:** mvn package
- **Purpose:**
  - Compiles code.
  - Runs tests.
  - Packages the application into a JAR/WAR in the target folder.
- **When to use:** When you want to create a distributable artifact.

7. Install the Artifact Locally

- **Command:** mvn install
- **Purpose:**
  - Does everything package does.
  - Installs (copies) the built artifact into your local Maven repository (~/.m2/repository).
- **When to use:** When you want to use this artifact as a dependency in another local project.

8. View Effective POM

- **Command:** mvn help:effective-pom
- **Purpose:** Shows the full, merged POM (your POM + inherited settings from parent POMs).
- **Use it for:** Understanding all the configuration and dependency versions in effect.

9. View Dependency Tree

- **Command:** mvn dependency:tree
- **Purpose:** Displays all direct and transitive dependencies in a tree structure.
- **Use it for:** Debugging dependency conflicts and understanding what libraries are included.

Summary Table

Command	What It Does
mvn --version	Shows Maven/Java/OS info
mvn compile	Compiles main code
mvn test-compile	Compiles main + test code
mvn clean	Deletes target folder (clean build)

mvn test	Runs unit tests
mvn package	Builds JAR/WAR
mvn install	Builds and installs to local repository
mvn help:effective-pom	Shows full effective POM
mvn dependency:tree	Shows dependency tree

Key Takeaways

- **Maven commands automate the entire build and test lifecycle.**
- Use the right command for the right job—compiling, testing, cleaning, packaging, or installing.
- Tools like dependency:tree and help:effective-pom help you understand and debug your project setup.
- Most commands can be run from the command line or through your IDE's Maven integration.

In summary:
Knowing these essential Maven commands will make you efficient and confident in building, testing, packaging, and managing dependencies for any Java project.

How Are Spring Releases Versioned? (Spring Versioning Scheme Explained)

1. Spring Version Format

- **Typical format:**
  major.minor.patch-modifier
  - **Example:** 6.1.3, 6.0.0-RC1, 5.3.0-M1
- **Modifier** is optional and indicates special versions (like milestones or release candidates).

2. What Do the Numbers Mean?

- **Major:**

    - Significant changes, possibly breaking backward compatibility.
    - Upgrading to a new major version (e.g., 4.x → 5.x, 5.x → 6.x) may require substantial code changes.
- **Minor:**
    - New features or enhancements, backward compatible.
    - Upgrading minor versions (e.g., 5.1 → 5.2) usually requires little or no code changes.
- **Patch:**
    - Bug fixes, security updates, small improvements.
    - Upgrading patch versions (e.g., 6.1.2 → 6.1.3) should require no code changes.

3. Modifiers: Special Version Suffixes

- **Milestone (M):**
    - Early preview, not production-ready.
    - Example: `6.0.0-M5` (Milestone 5)
- **Release Candidate (RC):**
    - Near-final, for last round of testing.
    - Example: `6.0.0-RC1`
- **Snapshot:**
    - Under development, unstable, should not be used in production.
    - Example: `6.0.0-SNAPSHOT`
- **Release:**
    - Stable, production-ready.
    - Example: `6.0.0` (no modifier means a final release)

4. Version Progression Example

1. **Snapshot:**
    - Ongoing development (`6.0.0-SNAPSHOT`)
2. **Milestones:**
    - Early access (`6.0.0-M1`, `6.0.0-M2`, ...)
3. **Release Candidates:**
    - Pre-release for testing (`6.0.0-RC1`, `6.0.0-RC2`)
4. **Release:**
    - Final version (`6.0.0`)

5. Where to Find These Versions?

- **Released versions:**
    - Available on Maven Central Repository
- **Milestones & RCs:**
    - Available on the Spring Milestone Repository
    - You'll need to add this repository in your `pom.xml` if you want to use milestone or RC versions.
- **Snapshots:**
    - Available on the Spring Snapshot Repository
    - Should not be used in production.

6. Recommendations

- **Avoid using SNAPSHOT versions**—they are unstable and not meant for production.
- **Use only released versions in production.**
- **Milestones and release candidates** are for testing, evaluation, or trying new features before the final release.

7. Quick Reference Table

Version Type	Example	Use Case	Where to find
Snapshot	6.0.0-SNAPSHOT	Development only	Snapshot Repo
Milestone (M)	6.0.0-M2	Early preview/test	Milestone Repo
Release Cand.	6.0.0-RC1	Last round testing	Milestone Repo
Release	6.0.0	Production	Maven Central

8. Summary

- **Spring uses a semantic versioning approach:**
  major.minor.patch-modifier
- **Modifiers** help distinguish between development (snapshot), early access (milestone), pre-release (RC), and final (release) versions.
- **Always prefer released (final) versions for production use.**

In summary:
Understanding the Spring versioning scheme helps you choose the right version for your project, plan upgrades, and avoid unstable builds in production.

Setting Goals for Long-Term Success

1. The Power of Long-Term Goals

- **Quote:**
  - *"People overestimate what they can do in a day and underestimate what they can achieve in a month."*
- **Lesson:**
  - Don't just focus on immediate tasks—think bigger and plan for the long term.

2. Example: The Instructor's Journey

- **Ambitious 3-Year Goal:**
  - Become recognized for expertise across all major cloud platforms (AWS, Azure, Google Cloud).
  - Become an expert in DevOps tools.
- **Why:**
  - Ambitious long-term goals provide direction and motivation.

3. Connecting Long-Term and Short-Term Goals

- **Short-Term Goals:**
  - Achievable tasks that contribute to long-term ambitions.
  - Example: "Deploy a full stack application to AWS, Azure, and Google Cloud."
- **Benefits:**
  - Short-term goals give a sense of accomplishment.
  - They act as stepping stones towards larger objectives.

4. Guidance for Learners

- **Set ambitious, long-term goals.**
- **Use those goals to guide your short-term actions and learning path.**
- **Track your progress by achieving milestones along the way.**

5. Motivation and Congratulations

- **Celebrate your milestones**—recognize the progress you're making.
- **Stay motivated** by always connecting your daily work to your bigger ambitions.

In summary:
Set bold long-term goals to guide your learning and career. Break these into meaningful, achievable short-term objectives. Celebrate each milestone, and remember that sustained, focused effort over time leads to significant achievements.

Getting Started with Spring Boot

21 July 2025 03:14

Getting Started with Spring Boot: Overview & Approach

1. Why Learn Spring Boot?

- **You can build web applications and REST APIs without Spring Boot,** but Spring Boot makes the process much simpler and faster.
- **Key Questions:**
  - Why is Spring Boot needed?
  - What problems does it solve compared to traditional Spring or Spring MVC?

2. Goals of Spring Boot

- Simplify the setup and development of Spring applications.
- Provide production-ready defaults and configurations.
- Reduce boilerplate code and manual configuration.

3. How Does Spring Boot Work?

- Uses **auto-configuration** to set up your application based on the libraries you have on the classpath.
- Relies on **starter projects** that bundle commonly used dependencies.
- Offers tools and features to improve developer productivity (e.g., Developer Tools, Actuator).

4. Comparing Frameworks

- **Spring:** The core framework, requires manual configuration.
- **Spring MVC:** Adds web and REST support, but still needs lots of setup.
- **Spring Boot:** Adds auto-configuration, starters, and tools to make everything easier.

5. Instructor's Experience

- The instructor has worked with Spring since 2005, and Spring Boot since its introduction in 2016.
- Will share insights into the challenges with traditional Spring and how Spring Boot addresses them.

6. Approach for This Section

1. **Understand the World Before Spring Boot**
   - Get a high-level overview of how applications were built using only Spring and Spring MVC.

2. **Create a Spring Boot Project**
    - Learn the modern, simplified approach.
3. **Build a Simple REST API Using Spring Boot**
    - See how little code and configuration is needed.
4. **Understand the "Magic" of Spring Boot**
    - Dive into key features:
        - **Spring Initializr:** Tool to quickly generate Spring Boot projects.
        - **Starter Projects:** Predefined dependency bundles for common use cases.
        - **Auto Configuration:** Automatic setup based on project needs.
        - **Developer Tools:** Enhancements for faster development.
        - **Actuator:** Production-ready features for monitoring and management.

7. Key Takeaways

- Spring Boot dramatically simplifies the process of building modern web and REST applications with Spring.
- The section will help you understand not just how to use Spring Boot, but why it exists and how it works under the hood.
- You'll compare the old way with the new, and learn about the powerful features Spring Boot provides.

In summary:
This section will guide you from understanding the challenges of traditional Spring development to building and managing applications efficiently with Spring Boot, exploring its features, and appreciating the productivity boost it offers.

The World Before Spring Boot: Challenges and Complexity

1. Setting Up Spring Projects Was NOT Easy

- **Many manual steps required** before a project was production-ready.
- **Lots of boilerplate and repetitive configuration** for every new project.

2. Major Challenges

a. Dependency Management

- Had to **manually add and manage all dependencies and their**

versions in pom.xml:
    - Spring core, Spring MVC, JSON libraries (e.g., Jackson), logging (e.g., log4j), etc.
- **Unit testing dependencies** (JUnit, Mockito, Spring Test) also had to be added and versioned.
- Example:

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-webmvc</artifactId>
  <version>6.2.2.RELEASE</version>
</dependency>
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <version>2.13.3</version>
</dependency>
<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>1.2.17</version>
</dependency>
```

b. Web Application Configuration (web.xml)

- Needed to **manually configure servlets**, such as Spring's DispatcherServlet.
- Example:

```
<servlet>
  <servlet-name>dispatcher</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/todo-servlet.xml</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>dispatcher</servlet-name>
  <url-pattern>/*</url-pattern>
</servlet-mapping>
```

c. Spring Configuration

- Had to **define beans, component scanning, and view resolvers** explicitly.
- Example:

```
<context:component-scan base-package="com.in28minutes" />
<bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
  <property name="prefix"><value>/WEB-INF/views/</value></property>
  <property name="suffix"><value>.jsp</value></property>
</bean>
```

d. Non-Functional Requirements (NFRs)

- **Logging, error handling, monitoring** had to be configured and coded manually.
- Example: Adding plugins and dependencies for logging and deploying to

servers.
- **Each project required redoing all this setup.**

3. The Result

- **Project setup took several days** for each new project.
- **Maintenance was time-consuming** and error-prone.
- **Repetitive manual work** for every application.

4. Summary Table

Challenge	Manual Steps Required
Dependency Management	Add/manage all dependencies and versions in pom.xml
Web Configuration	Edit web.xml for servlets and mappings
Spring Beans & Components	Define beans, component scan, view resolvers, etc.
Non-Functional Requirements	Logging, monitoring, error handling setup
Repeat For Every Project	Yes

5. Looking Forward

- **Spring Boot automates and simplifies** all of these steps.
- Next: You'll see how Spring Boot makes setup and development dramatically easier.

In summary:
Before Spring Boot, Java developers spent a lot of time configuring dependencies, web.xml, Spring beans, and NFRs for every new project. Spring Boot was created to eliminate this repetitive, error-prone work and streamline the process of building modern applications.

Step-by-Step: Creating a Spring Boot Project with Spring Initializr

1. Using Spring Initializr

- **Spring Initializr Website:**
  - Visit start.spring.io to quickly set up a new Spring Boot project.
- **Simple Choices to Make:**
  - **Build Tool:** Maven (recommended for this course)
  - **Language:** Java
  - **Spring Boot Version:**
    - Choose a release version (e.g., 3.1.0, 3.2.0).
    - If only milestone versions (like 3.0.0-M3) are available, use the latest one.

        - **Avoid SNAPSHOT versions** (they are under development and unstable).
    - **Project Metadata:**
        - **Group ID:** Like a package name (e.g., com.in28minutes.springboot)
        - **Artifact ID:** Like a project/class name (e.g., learn-spring-boot)
    - **Java Version:** Spring Boot 3 requires Java 17 or higher.
    - **Dependencies:**
        - For a REST API, add **Spring Web** (also called Spring Boot Starter Web).

2. Generating and Importing the Project

- **Generate:**
    - Click the "Generate" button to download a zip file containing your new project.
- **Extract:**
    - Unzip the downloaded project to a convenient folder.
- **Import into Eclipse:**
    - Open Eclipse (preferably the latest version for Java Enterprise).
    - Use **File → Import → Maven → Existing Maven Projects**.
    - Browse to your extracted folder, select it, and finish the import.
    - The import may take 10–15 minutes the first time (downloads dependencies).

3. Project Structure

- **src/main/java:** Main Java source code.
- **src/main/resources:** Application configuration and resources.
- **src/test/java:** Unit tests.
- **pom.xml:** Project dependencies and configuration.
- **Maven Dependencies:** Managed automatically (you'll see lots of them in your project).

4. Launching the Application

- **Find the Main Class:**
    - In src/main/java, locate the package you specified (e.g., com.in28minutes.springboot.learnspringboot).
    - Open the main class (e.g., LearnSpringBootApplication.java).
- **Run:**
    - Right-click the main class → Run As → Java Application.
    - Spring Boot will start an embedded Tomcat server (usually on port 8080).
    - The application starts up quickly, and logs are printed to the console.

5. Key Points

- **Spring Initializr** makes it extremely easy to start a new Spring Boot project—just a few clicks!
- **Maven** handles all dependencies for you (thanks to starter projects like Spring Web and Spring Boot Starter Test).
- **Launching** the app is as simple as running the main method—no extra

configuration needed.
- **Embedded Tomcat** means no need to deploy to an external server for development.

6. What's Next?

- The next step will show you how to **build a simple REST API** using Spring Boot, demonstrating how quick and easy it is.

In summary:
Creating a Spring Boot project is quick and straightforward with Spring Initializr. You define a few settings, add needed dependencies, import into your IDE, and run it. Spring Boot's opinionated defaults and embedded server make setup and first launch fast and painless.

Building a Simple REST API with Spring Boot

1. Objective

- **Goal:** Demonstrate how easy it is to create a REST API using Spring Boot.
- **Target:** Expose a `/courses` endpoint that returns a list of course objects as JSON.

2. Steps in the Lecture (and Code Explanation)

a. Creating the Controller

- **Lecture:**
  - Create a new class called `CourseController`.
  - Mark it with `@RestController` annotation.
  - Define a method to handle HTTP requests and return a list of courses.
- **Code:**

```
@RestController
public class CourseController {
    @RequestMapping("/courses")
    public List<Course> retrieveAllCourses() {
        return Arrays.asList(
            new Course(1, "Learn AWS", "in28minutes"),
            new Course(2, "Learn DevOps", "in28minutes")
        );
    }
}
```

- **Explanation:**
  - `@RestController` tells Spring Boot to treat this class as a REST API controller.
  - `@RequestMapping("/courses")` maps HTTP GET requests to `/courses` to the method.

- The method returns a List<Course>, which Spring Boot automatically serializes to JSON.

b. Creating the Course Bean

- **Lecture:**
  - Create a Course class with id, name, and author fields.
  - Generate a constructor, getters, and a toString() method.
- **Code:**

```
public class Course {
    private long id;
    private String name;
    private String author;
    // Constructor, getters, toString()
}
```

- **Explanation:**
  - This is a simple POJO ("Plain Old Java Object") used to represent course data.
  - Getters are required so Spring can serialize the fields into JSON.

c. Running the Application

- **Lecture:**
  - Run the main application class (LearnSpringBootApplication) as a Java application.
  - Visit http://localhost:8080/courses in your browser.
- **Code:**

```
@SpringBootApplication
public class LearnSpringBootApplication {
    public static void main(String[] args) {
        SpringApplication.run(LearnSpringBootApplication.class, args);
    }
}
```

- **Explanation:**
  - @SpringBootApplication is a convenience annotation that sets up everything for a Spring Boot app.
  - SpringApplication.run(...) launches the embedded web server (Tomcat by default).

d. Result

- **Lecture:**
  - Visiting /courses returns a JSON array of courses.
  - If you want better formatting, use a JSON formatter browser extension.
- **Your Output:**

```
[
  { "id": 1, "name": "Learn AWS", "author": "in28minutes" },
  { "id": 2, "name": "Learn DevOps", "author": "in28minutes" }
]
```

- **Explanation:**
  - Spring Boot automatically converts the list of Course objects to JSON and returns it in the response.
  - No need for any extra configuration, XML, or bean setup—Spring Boot

handles it all behind the scenes.

3. Key Takeaways

- **Simplicity:**
  - You only need to write the business logic and annotate your classes/methods.
  - No XML configuration, no manual servlet or bean setup.
- **Convention over Configuration:**
  - Spring Boot's auto-configuration and sensible defaults do all the heavy lifting.
- **Fast Feedback:**
  - You can see your REST API running with just a few lines of code.
- **Modern Development:**
  - Embedded server (Tomcat), automatic JSON serialization, minimal setup.

4. What's Next?

- The next steps in the course will explain **how Spring Boot makes this possible** and what happens behind the scenes to enable such a streamlined experience.

In summary:
With Spring Boot, creating a REST API is quick and straightforward. Write your business logic, use the right annotations, and Spring Boot takes care of the rest—getting you productive and running faster than ever before.

Full Code:

LearnSpringBootApplication.java

```
package com.in28minutes.springboot.learn_spring_boot;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class LearnSpringBootApplication {

	public static void main(String[] args) {
		SpringApplication.run(LearnSpringBootApplication.class, args);
	}

}
```

CourseController.java

```
package com.in28minutes.springboot.learn_spring_boot;

import java.util.Arrays;
import java.util.List;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class CourseController {
```

```
	@RequestMapping("/courses")
	public List<Course> retrieveAllCourses() {
		return Arrays.asList(
				new Course(1, "Learn AWS", "in28minutes"),
				new Course(2, "Learn DevOps", "in28minutes")


				);
	}
}
```

Course.java

```
package com.in28minutes.springboot.learn_spring_boot;

public class {
	private long id;
	private String name;
	private String author;

	public Course(long id, String name, String author) {
		super();
		this.id = id;
		this.name = name;
		this.author = author;
	}

	public long getId() {
		return id;
	}

	public String getName() {
		return name;
	}

	public String getAuthor() {
		return author;
	}

	@Override
	public String toString() {
		return "Course [id=" + id + ", name=" + name + ", author=" + author
+ "]";
	}

}
```

http://localhost:8080/courses

What's the Most Important Goal of Spring Boot?

1. The Core Goal

- **Help you build production-ready apps quickly**
  - **Quickly:**
    - Rapid application development.
    - Minimize setup and boilerplate.
  - **Production-Ready:**
    - Apps are ready to be deployed in real-world environments from the start.
    - Includes essential features (like logging, monitoring, configuration management).

2. Features for Building Quickly

- **Spring Initializr:**
  - start.spring.io
  - Quickly generate a new Spring Boot project with required dependencies.
- **Spring Boot Starter Projects:**
  - Predefined dependency bundles for common use cases (web, data, security, etc.).
  - Reduces effort in managing individual dependencies and versions.
- **Spring Boot Auto Configuration:**
  - Automatically configures the application based on the dependencies on the classpath.
  - Removes the need for most manual configuration.
- **Spring Boot DevTools:**
  - Enhances developer productivity.
  - Enables features like automatic restarts and live reload during development.

3. Features for Being Production-Ready

- **Logging:**
  - Built-in support for logging with sensible defaults.

- **Environment-Specific Configuration:**
  - Use different settings for dev, test, staging, and production environments.
  - Achieved via **Profiles** and **ConfigurationProperties**.
- **Monitoring:**
  - **Spring Boot Actuator** provides endpoints for health checks, metrics, and monitoring.
  - Helps you observe application health and usage in production.

4. Why Both "Quickly" and "Production-Ready" Matter

- It's not enough to build apps fast—they must also be robust, maintainable, and ready for real-world deployment.
- Spring Boot bridges the gap:
  - **Fast to build.**
  - **Easy to maintain and monitor.**

5. Key Takeaways

- **Spring Boot's #1 goal:**
  - Enable rapid development of applications that are immediately ready for production.
- **Critical features:**
  - Spring Initializr, Starters, Auto Configuration, DevTools (for speed).
  - Logging, profiles, configuration management, Actuator (for production-readiness).
- **Next Steps:**
  - Explore each of these features in detail, starting with Spring Boot Starter Projects.

In summary:
Spring Boot's most important goal is to let you build production-ready applications rapidly, by providing both fast development tools and built-in production features out of the box.

Exploring Spring Boot Starter Projects

1. Why Starters?

- **Building modern applications requires many frameworks:**
  - For a REST API: Spring Core, Spring MVC, Tomcat (server), JSON converter, etc.
  - For unit testing: Spring Test, JUnit, Mockito, etc.
- **Manually listing and versioning all required dependencies is tedious and error-prone.**

2. What Are Spring Boot Starter Projects?

- **Starters are convenient dependency descriptors**—predefined bundles of dependencies for specific functionalities.
- **Purpose:**
  - Group all required libraries for a particular feature (like web, testing, database, etc.) into a single dependency.
  - Simplify your pom.xml—add just one starter, and get everything you need.
- **How it works:**
  - When you add a starter dependency in your pom.xml, Maven pulls in all the necessary libraries automatically.

3. Examples of Common Starters

Use Case	Starter Dependency	What It Brings In
Web/REST API	spring-boot-starter-web	Spring MVC, Spring Web, Tomcat, JSON converter (e.g., Jackson)
Unit Testing	spring-boot-starter-test	JUnit, Mockito, Spring Test, AssertJ, Hamcrest, etc.
Database (JPA)	spring-boot-starter-data-jpa	Spring Data JPA, Hibernate, etc.
Database (JDBC)	spring-boot-starter-jdbc	Spring JDBC, connection pool, etc.
Security	spring-boot-starter-security	Spring Security and its dependencies

4. How Starters Work in Practice

- **In your pom.xml:**

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
```

- **This automatically brings in all the libraries needed for web development and testing.**

5. Key Takeaways

- **Spring Boot Starters simplify project setup:**
  - No more hunting for individual dependencies or worrying about version mismatches.
- **Starters cover a wide range of features:**
  - Pick the starter that matches your use case (web, data, security, etc.).
- **They make your project's dependencies clean, consistent, and easy to manage.**

6. What's Next?

- **Having the right dependencies is just step one.**

- **Next:** Spring Boot's auto-configuration kicks in to wire everything up automatically—no need for manual configuration. This will be covered in the following lesson.

In summary:
Spring Boot Starter Projects are predefined collections of dependencies for different features (web, testing, data, security, etc.). By adding the right starter to your project, you instantly get all the libraries you need, making setup fast, easy, and reliable.

Exploring Spring Boot Auto Configuration

1. The Challenge: Manual Configuration in Spring

- **Traditional Spring applications require lots of configuration:**
  - Component scanning
  - DispatcherServlet setup
  - DataSource (for databases)
  - JSON conversion (bean ↔ JSON)
  - Error pages
  - Web server setup (Tomcat, Jetty, etc.)
- **Problem:** Tedious, repetitive, and error-prone for every project

2. Spring Boot Solution: Auto Configuration

- **Auto Configuration = Automated configuration for your app**
  - Spring Boot configures your application automatically, based on:
    - **What frameworks/libraries are on the classpath** (e.g., if you have Spring Web, JPA, Security, etc.)
    - **What configuration you've already provided** (e.g., annotations, application.properties)
- **You get sensible defaults** for most use cases, but can override them if needed.

3. How Does Auto Configuration Work?

- **Spring Boot Starter Projects** (like spring-boot-starter-web) bring necessary libraries into your project.
- **Spring Boot scans the classpath** for these libraries and applies matching auto-configurations.
- **Auto-configuration logic is packaged in** spring-boot-autoconfigure.jar.

4. Key Examples (from Starters and Auto-Config Classes)

Feature	Auto-Config Class	What It Does
DispatcherServlet	DispatcherServletAutoConfiguration	Sets up the central web controller
Embedded Web Server	EmbeddedWebServerFactoryCustomizerAutoConfiguration	Configures Tomcat (default), Jetty, etc. as the embedded server
Error Pages	ErrorMvcAutoConfiguration	Sets up a default error page (WhiteLabel error page)
JSON Conversion	JacksonHttpMessageConvertersConfiguration	Enables automatic bean ↔ JSON conversion via Jackson

5. How to Explore Auto Configuration

- **Look at your dependencies:**
  - E.g., adding spring-boot-starter-web brings in Spring MVC, Tomcat, and Jackson.
- **Look inside spring-boot-autoconfigure.jar:**
  - Contains many *AutoConfiguration classes, each handling a specific feature.
- **Customize via application.properties:**
  - Example: Change logging level for debugging auto-configuration.
    `logging.level.org.springframework=debug`
  - Spring Boot will use your custom settings over the defaults.

6. How Spring Boot Decides What to Auto-Configure

- **Which frameworks are in the classpath?**
  - E.g., if Spring Web is present, configure DispatcherServlet and Tomcat.
- **What existing configuration is present?**
  - If you provide your own beans or settings, Spring Boot respects them and does not override.

7. Practical Example

- **You add spring-boot-starter-web:**
  - **Spring Boot automatically configures:**
    - DispatcherServlet (handles web requests)
    - Embedded Tomcat server (runs your app)
    - Error pages (like the WhiteLabel error page for unmapped URLs)
    - JSON converters (Jackson) for bean ⇔ JSON conversion

8. Key Takeaways

- **Spring Boot Auto Configuration saves you from writing repetitive setup code.**
- **It works by inspecting your project's dependencies and existing configuration.**
- **You can override any default by providing your own configuration.**
- **All auto-configuration logic is in spring-boot-autoconfigure.jar.**

In summary:
Spring Boot's Auto Configuration is a powerful feature that automatically sets up

Build Faster with Spring Boot DevTools

In summary:
Spring Boot's auto-configuration feature automatically sets up your application based on what's present in your project. This drastically reduces manual configuration and makes getting started with Spring Boot fast and easy.

Build Faster with Spring Boot DevTools

1. What is Spring Boot DevTools?

- **A tool to improve developer productivity** when building Spring Boot applications.
- **Main goal:**
  - Reduce time and effort during development by eliminating the need for manual server restarts on most code changes.

2. Why is it Useful?

- **Typical Problem:**
  - In traditional development, every code change (Java, properties, templates) usually requires a manual server restart to see the effect.
  - This slows down the feedback loop and productivity.
- **Spring Boot DevTools Solution:**
  - Automatically detects code or resource changes and restarts the application for you.

3. How to Use DevTools

- **Add the Dependency:**
  - In your `pom.xml`, add:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <optional>true</optional>
</dependency>
```

- **Restart Behavior:**
  - When you edit and save Java files, property files, or static resources, DevTools triggers an automatic application restart.
  - Changes are picked up instantly—just refresh your browser to see the effect.
  - **Example:**
    - Add a new course to your controller, save, and the new course appears on `/courses` after an automatic restart.

4. Important Limitations

- **For changes in `pom.xml` (dependency changes):**
  - You **must restart the server manually.**
  - DevTools cannot reload changes in project dependencies or build plugins.
- **For all other files (Java, properties, templates, resources):**
  - DevTools handles automatic restarts.

5. Key Benefits

- **Faster feedback loop:**
  - See your changes instantly, without manual restarts.

- **Increases productivity:**
  - Less waiting, more coding and testing.
- **Easy to use:**
  - Just add the dependency—no extra setup.

6. Recap Table

File Changed	Do You Need to Restart Manually?
Java/Class/Properties	No (DevTools does it)
Static Resources	No (DevTools does it)
pom.xml/Dependencies	Yes (restart needed)

7. Summary

- **Spring Boot DevTools** is a must-have for efficient Spring Boot development.
- It automates application restarts for most changes, streamlining your workflow.
- Remember: Manual restarts are only needed for dependency or build configuration changes.

In summary:
Spring Boot DevTools accelerates your development process by automatically restarting your application whenever you make most types of changes, giving you a smoother and faster feedback cycle.

```
package com.in28minutes.springboot.learn_spring_boot;

import java.util.Arrays;
import java.util.List;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class CourseController {

	@RequestMapping("/courses")
	public List<Course> retrieveAllCourses() {
		return Arrays.asList(
				new Course(1, "Learn AWS", "in28minutes"),
				new Course(2, "Learn DevOps", "in28minutes"),

				new Course(3, "Learn Azure", "in28minutes"),
				new Course(4, "Learn GCP", "in28minutes")

				);
	}
}
```

```
[
  {
    "id": 1,
    "name": "Learn AWS",
    "author": "in28minutes"
  },
  {
    "id": 2,
    "name": "Learn DevOps",
    "author": "in28minutes"
  },
  {
    "id": 3,
    "name": "Learn Azure",
    "author": "in28minutes"
  },
  {
    "id": 4,
    "name": "Learn GCP",
    "author": "in28minutes"
  }
]
```


Spring Boot: Managing Application Configuration Using Profiles

1. Why Do We Need Profiles?

- **Applications run in different environments:**
  - Development (Dev)
  - Quality Assurance (QA)
  - Staging
  - Production (Prod)
- **Each environment often requires different configuration:**
  - Example: Different databases or web service endpoints for Dev, QA, and Prod.
- **Hard-coding configuration for every environment is error-prone and difficult to maintain.**

2. What Are Profiles?

- **Profiles** are a Spring Boot feature that lets you define environment-specific configurations.
- **How it works:**
  - You create separate property files for each environment, e.g.:

        - application-dev.properties
        - application-prod.properties
        - application-qa.properties
    - Each file contains the settings specific to that environment.

3. How to Use Profiles

- **Default configuration:**
    - application.properties contains properties used when no profile is active.
- **Profile-specific configuration:**
    - Add files like application-dev.properties or application-prod.properties in src/main/resources.
    - Example contents:
        - application-dev.properties:
          logging.level.org.springframework=trace
        - application-prod.properties:
          logging.level.org.springframework=info
- **Activating a profile:**
    - In application.properties, set:
      spring.profiles.active=dev
    - This tells Spring Boot to use both application.properties and application-dev.properties. If a property exists in both, the profile-specific one wins.

4. Example: Logging Levels

- **Different environments may require different logging:**
    - Dev: Most detailed logs (trace)
    - QA: Moderate logs (debug or info)
    - Prod: Minimal logs (info or warning)
- **Log levels (from most to least verbose):**
    - trace > debug > info > warning > error > off
- **How they work:**
    - If you set trace, all log messages (trace, debug, info, warning, error) will be shown.
    - If you set info, only info, warning, and error messages are shown.

5. Example Code (from your config)

- **application.properties:**
  spring.application.name=learn-spring-boot
  logging.level.org.springframework=debug
  spring.profiles.active=dev
- **application-dev.properties:**
  logging.level.org.springframework=trace
- **application-prod.properties:**
  logging.level.org.springframework=info
- **Behavior:**
    - With spring.profiles.active=dev, logging.level.org.springframework is set to trace (overrides debug from the default).
    - If you change to spring.profiles.active=prod, logging.level.org.springframework is set to info.

6. Key Takeaways

- **Profiles** allow you to cleanly separate configuration for different environments.
- **Profile-specific property files** override the defaults, making deployment safer and easier.
- **Logging levels** and other environment-specific settings are easy to manage and switch.
- **Profiles are just the start:** For more advanced configuration, Spring Boot also supports @ConfigurationProperties for mapping groups of related properties into Java beans.

In summary:
Spring Boot Profiles are essential for managing environment-specific configurations. By using different application-<profile>.properties files and activating the right profile, you ensure your app always uses the correct settings for each environment, improving flexibility, safety, and maintainability.

application.properties:
```
spring.application.name=learn-spring-boot
logging.level.org.springframework=debug
spring.profiles.active=dev
```

application-dev.properties:
```
spring.application.name=learn-spring-boot
logging.level.org.springframework=trace
```

application-prod.properties:
```
spring.application.name=learn-spring-boot
logging.level.org.springframework=info
```

Spring Boot: Externalized & Environment-Specific Configuration with @ConfigurationProperties and Profiles

1. The Problem: Managing Complex, Environment-Specific Configuration

- Real-world apps need different settings for each environment (Dev, QA, Prod, etc.)
- You may have many related properties for services (e.g., currency-service URL, username, key)
- Hard-coding or scattering these in code is error-prone and hard to maintain

2. The Solution: @ConfigurationProperties

- **Purpose:**
  - Group related configuration properties into a single Java class

- Bind values from application.properties (or application-<profile>.properties) directly to fields

- **How to use:**
  1. **Create a POJO:**
     - Example: CurrencyServiceConfiguration
     - Fields for url, username, key
  2. **Annotate with @ConfigurationProperties(prefix = "currency-service")**
     - Spring will map all properties starting with currency-service. to this class
  3. **Annotate with @Component**
     - So Spring manages and injects the class as a bean
  4. **Add getters and setters** for all fields (required for property binding)

```
@ConfigurationProperties(prefix = "currency-service")
@Component
public class CurrencyServiceConfiguration {
   private String url;
   private String username;
   private String key;
   // Getters and setters...
}
```

3. Using the Configuration in Your Controller

- **Inject the configuration bean** into your controller using @Autowired
- **Expose it via a REST endpoint** (e.g., /currency-configuration) to see the current settings

```
@RestController
public class CurrencyConfigurationController {
   @Autowired
   private CurrencyServiceConfiguration configuration;

   @RequestMapping("/currency-configuration")
   public CurrencyServiceConfiguration retrieveAllCourses() {
      return configuration;
   }
}
```

4. Defining Properties for Different Environments (Profiles)

- **application.properties** (default for all environments)
- **application-dev.properties** (for dev)
- **application-prod.properties** (for prod)
- **Activate a profile** with:
  spring.profiles.active=dev
- **Example properties:**
  - In application.properties:
    C ,.lknurrency-service.url=http://default1.in28minutes.com
    currency-service.username=defaultusername
    currency-service.key=defaultkey

- In application-dev.properties:
  currency-service.url=http://dev.in28minutes.com
  currency-service.username=devusername
  currency-service.key=devkey

5. Behavior and Testing

- **Spring Boot will use the active profile's property file** to populate the configuration bean.
- **If a property is not set in the active profile, it falls back to the default file.**
- **Changing the profile and refreshing the endpoint** (/currency-configuration) will show the values for that environment.

6. Common Pitfalls

- **Spelling matters:**
  - In your properties you had currency-serivce.username (typo) — should be currency-service.username to match the prefix!
- **All properties must match the prefix in @ConfigurationProperties.**

7. Key Takeaways

- **@ConfigurationProperties** makes it easy to manage groups of related settings, reducing clutter and making configuration type-safe.
- **Profiles** allow you to easily switch configuration for different environments without code changes.
- **Combining both** is a best practice for scalable, maintainable Spring Boot applications.

In summary:
Use @ConfigurationProperties to bind groups of related configuration values to Java classes, and use Spring Boot profiles to externalize and override configuration per environment. This approach keeps your application flexible, maintainable, and production-ready.

application.properties:

```
spring.application.name=learn-spring-boot
logging.level.org.springframework=debug
spring.profiles.active=dev


currency-service.url=http://default1.in28minutes.com
currency-serivce.username=defaultusername
currency-service.key=defaultkey
```

application-dev.properties:

```
spring.application.name=learn-spring-boot
logging.level.org.springframework=trace

currency-service.url=http://dev.in28minutes.com
currency-serivce.username=devusername
currency-service.key=devkey
```

```
CurrencyServiceConfiguration.java
package com.in28minutes.springboot.learn_spring_boot;
```

```
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.stereotype.Component;

// currency-service.url=
// currency-serivce.username=
// currency-service.key=

@ConfigurationProperties(prefix = "currency-service")
@Component
public class {

	private String url;
	private String username;
	private String key;

	public String getUrl() {
		return url;
	}

	public void setUrl(String url) {
		this.url = url;
	}

	public String getUsername() {
		return username;
	}

	public void setUsername(String username) {
		this.username = username;
	}

	public String getKey() {
		return key;
	}

	public void setKey(String key) {
		this.key = key;
	}

}
```

CurrencyConfigurationController.java

```
package com.in28minutes.springboot.learn_spring_boot;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class CurrencyConfigurationController {

	@Autowired
	private CurrencyServiceConfiguration configuration;

	@RequestMapping("/currency-configuration")
	public CurrencyServiceConfiguration retrieveAllCourses() {
		return configuration;
	}
```

}

Simplify Deployment with Spring Boot Embedded Servers

1. Traditional ("WAR") Deployment Approach

- **Steps:**
  1. **Install Java** on the server.
  2. **Install a web/application server** (e.g., Tomcat, WebSphere, WebLogic).
  3. **Deploy the application as a WAR (Web ARchive) file** to the server.
- **Drawbacks:**
  - Multiple manual setup steps.
  - Server maintenance and configuration overhead.
  - More complex, especially across multiple environments (dev, QA, staging, prod).

2. The Embedded Server Approach (Spring Boot Way)

- **How it works:**
  - The server (e.g., Tomcat) is **embedded inside your application's JAR file.**
  - No need to separately install and configure a web server.
- **Deployment Steps:**
  1. **Install Java** on the machine.
  2. **Run your JAR file** with a single command:
     java -jar your-app.jar
- **Benefits:**
  - **Much simpler deployment.**
  - Consistency across all environments.
  - Makes automation and cloud deployment easier.
  - "Make JAR not WAR!" (Josh Long, Spring advocate)

3. How It Works in Spring Boot

- **Spring Boot Starters** bring in the embedded server:
  - spring-boot-starter-web includes **Tomcat** by default.
  - You can switch to **Jetty** or **Undertow** by adding:
    - spring-boot-starter-jetty
    - spring-boot-starter-undertow
- **No need for separate web server installation.**

- **Building your app** (e.g., with mvn clean install) creates a runnable JAR containing both your code and the server.

4. Practical Example

- **Build the app:**
  Use Maven or Gradle to build your Spring Boot app:
  mvn clean install
- **Find the JAR:**
  Look in the target/ folder for something like learn-spring-boot-0.0.1-SNAPSHOT.jar.
- **Run the app:**
  java -jar learn-spring-boot-0.0.1-SNAPSHOT.jar
- **That's it!**
  Your application and server are up and running.

5. Supported Embedded Servers

Embedded Server	Starter Dependency
Tomcat (default)	spring-boot-starter-tomcat
Jetty	spring-boot-starter-jetty
Undertow	spring-boot-starter-undertow

6. Key Takeaways

- **Embedded servers** make deployments faster, simpler, and more consistent.
- **Only Java is required** on the deployment machine—no separate server setup.
- This model is highly compatible with **modern DevOps, cloud, and container** environments.

In summary:
Spring Boot's embedded server approach means you just need Java and your JAR file to run your app—no more complicated WAR deployments or separate server installations. This greatly simplifies and speeds up the deployment process for modern applications.

Monitor Applications Using Spring Boot Actuator

1. Why Monitoring Matters

- **Production-ready applications** need to be monitored and managed.
- Monitoring helps you:
  - Understand what's happening inside your app.
  - Diagnose issues and ensure uptime.
  - Track resource usage and health.

2. What is Spring Boot Actuator?

- A **Spring Boot feature** for monitoring and managing your application.
- **Exposes a variety of endpoints** (URLs) that provide detailed information about the app and its environment.

3. Key Actuator Endpoints

Endpoint	What it Shows
/actuator/beans	Complete list of Spring beans in your app (useful for debugging auto-configuration)
/actuator/health	Application health info (e.g., status: UP/DOWN)
/actuator/metrics	Application and system metrics (JVM, memory, HTTP requests, etc.)
/actuator/mappings	Details about all request mappings (controllers, endpoints)

Other endpoints include: /actuator/configprops, /actuator/env, etc.

4. How to Enable and Use Actuator

- **Add the dependency** to your pom.xml:
  ```
  <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-actuator</artifactId>
  </dependency>
  ```
- **Run your application.**
- **Access endpoints:**
  - By default, only /actuator/health is enabled.

  - To enable all endpoints, add to application.properties:
    management.endpoints.web.exposure.include=*
  - Or, expose only specific endpoints (recommended for production):
    management.endpoints.web.exposure.include=health,metrics

5. What You Can See and Do

- **/actuator/beans**: View all beans loaded in the Spring context (great for

debugging what's auto-configured).
- **/actuator/health**: Instantly check if your app is UP and see details about dependencies (DB, disk space, etc.).
- **/actuator/metrics**: View metrics like JVM memory, disk usage, HTTP request counts and times, etc.
    - Drill down: /actuator/metrics/http.server.requests shows request counts, timings, etc.
- **/actuator/mappings**: See all routes/endpoints in your app and what methods handle them.
- **/actuator/configprops**: View all configuration properties and their values.
- **/actuator/env**: See environment variables, system properties, and Spring environment info.

6. Best Practices & Notes

- **Expose only necessary endpoints** in production (for security and performance).
- **Actuator can consume resources**, exposing many endpoints increases monitoring overhead.
- Integrates well with monitoring tools and cloud platforms.

7. Key Takeaways

- **Spring Boot Actuator** provides out-of-the-box monitoring and management endpoints.
- Enables you to **monitor application health, metrics, beans, request mappings, and more**.
- **Easy to enable and configure**; essential for building production-grade applications.

In summary:
Spring Boot Actuator is a powerful tool to monitor and manage your Spring Boot application, giving you instant insight into its health, configuration, and performance with minimal setup.

Understanding Spring Boot vs Spring MVC vs Spring Framework

1. Spring Framework: The Foundation

- **Core Focus:**
    - **Dependency Injection (DI):**
        - Manages and wires dependencies between Java objects.
        - Uses annotations like @Component, @Service, @Autowired.
        - Relies on **component scan** to discover beans and wire them together.
- **Limitations:**
    - **DI alone isn't enough** for real-world apps.
    - Need additional frameworks for:
        - Database access (Hibernate, JPA)
        - Unit testing (JUnit, Mockito)
- **Spring Modules and Projects:**

- Extend the ecosystem and provide smooth integration with other tools.

2. Spring MVC: The Web Module

- **What is it?**
  - A module of the Spring Framework focused on **building web applications and REST APIs**.
- **Key Features:**
  - Simplifies web app and REST API development compared to older frameworks like Struts.
  - Uses annotations like:
    - @Controller
    - @RestController
    - @RequestMapping("/courses")
- **Scope:**
  - Only covers the web layer (controllers, request handling, etc.).

3. The Challenge Before Spring Boot

- Even with Spring and Spring MVC, **configuration was complex**:
  - Many XML files (pom.xml, web.xml, applicationContext.xml)
  - Tedious and repetitive setup for each new project
  - Hard to get "production-ready" quickly

4. Spring Boot: The Game Changer

- **What is it?**
  - A Spring project that builds on top of Spring and Spring MVC.
- **Primary Goal:**
  - **Help you build production-ready applications quickly.**
- **Key Features:**
  - **Starter Projects:**
    - Bundled dependencies for common use cases (web, data, security, etc.).
  - **Auto Configuration:**
    - Automatically configures your app based on the classpath and minimal setup.
    - Eliminates boilerplate and XML configuration.
  - **Non-Functional Requirements (NFRs):**
    - **Actuator:** Advanced monitoring and management endpoints.
    - **Embedded Server:** No need for external Tomcat or WebLogic—just run your JAR.
    - **Logging & Error Handling:** Sensible defaults out of the box.
    - **Profiles & ConfigurationProperties:** Easy environment-specific and externalized configuration.

5. Relationship Between the Three

- **Spring Boot is not a competitor to Spring or Spring MVC.**
  - It is a **wrapper** that makes using Spring and Spring MVC easier and faster.
  - It brings together best practices, defaults, and automation so you can focus on business logic.

6. Summary Table

Feature/Layer	Spring Framework	Spring MVC	Spring Boot
Dependency Injection	Yes	Yes (inherits)	Yes (inherits)
Web/REST APIs	No	Yes	Yes (via MVC)
Starter Projects	No	No	Yes
Auto Configuration	No	No	Yes
Embedded Server	No	No	Yes
Actuator/Monitoring	No	No	Yes
Profiles/ConfigProps	No	No	Yes

7. Key Takeaways

- **Spring Framework:** Core DI and ecosystem integration.
- **Spring MVC:** Builds on Spring for web and REST APIs.
- **Spring Boot:** Makes Spring and Spring MVC easy, fast, and production-ready through starters, auto-configuration, embedded server, monitoring, and simplified configuration.

In summary:
Spring Boot sits on top of Spring and Spring MVC, wrapping them with powerful features that drastically reduce configuration and setup time, letting you build and deploy robust, production-ready applications with minimal fuss.

Spring Boot – Section Review & 10,000 Feet Overview

1. Section Purpose

- **Goal:**
  - Provide a **high-level (10,000 feet) overview of Spring Boot**
  - Help you understand key terminology and concepts foundational to Spring Boot development

2. Key Spring Boot Terminology & Features Reviewed

- **Starter Projects:**
  - Pre-configured dependency bundles for common use cases (web, data, security, etc.)
  - Make project setup fast and easy
- **Auto Configuration:**
  - Automatically configures your application based on dependencies and minimal code
  - Eliminates repetitive boilerplate and manual setup
- **Actuator:**
  - Provides built-in endpoints for monitoring and managing your application (health, metrics, beans, etc.)
  - Essential for production readiness
- **DevTools:**
  - Improves developer productivity by enabling automatic restarts and live reloads during development
  - Fast feedback loop for code changes

3. Main Advantage of Spring Boot

- **Get started quickly with production-ready features:**
  - Out-of-the-box support for monitoring, configuration, logging, error handling, and deployment
  - Drastically reduces time and effort to build robust, maintainable applications

4. Looking Ahead

- The foundation you've built in understanding these concepts will be **invaluable as you start building more features and go deeper into Spring Boot** in the next sections.

5. Key Takeaways

- **Spring Boot is designed to accelerate development** by providing smart defaults, automation, and essential production features.
- **Understanding the terminology and core features** (starters, auto-configuration, actuator, DevTools) is crucial for effective Spring Boot development.

In summary:
This section gave you a broad overview of Spring Boot's core features and terminology, setting you up to confidently dive into hands-on development and more advanced topics.

Congratulations & The Origin of in28minutes

1. Congratulations & Encouragement

- **Milestone Achieved:**
  - Recognition of your progress in the course.
  - Emphasis on the value of investing in your own learning and growth.

2. The Story Behind "in28minutes"

- **Common Learner Question:**
  - Many students ask, “What is in28minutes? Where does the name come from?”
- **Origin Story:**
  - Inspired by the instructor’s experiences working with talented people across India, the US, and Europe.
  - **Key characteristics of successful people:**
    - **Embrace Change:**
      - See new changes (like the rise of cloud computing) as opportunities to learn, not threats.
    - **Inquisitiveness:**
      - Always ask “why” and seek to understand the reasons behind decisions and events.
    - **Continuous Learning:**
      - Dedicate time each day to learning something new—both at work and outside work.
  - **The Name “in28minutes”:**
    - Represents the commitment to learning something new every day.
    - Specifically, dedicating at least 28 minutes daily to learning and self-improvement.

3. Key Takeaways

- **Mindset for Success:**
  - See change as an opportunity.
  - Stay curious and keep questioning.
  - Make continuous learning a daily habit.
- **Encouragement:**
  - The journey of learning and self-investment is ongoing and rewarding.
  - The course and the community are here to support your growth.

In summary:
The name "in28minutes" reflects the philosophy of daily, consistent learning and growth. By staying curious, embracing change, and making learning a habit, you set yourself up for ongoing success—both in technology and in life.

Getting Started with JPA and Hibernate with Spring and Spring Boot

26 July 2025 23:54

JPA and Hibernate in 10 Steps: Section Overview

1. Section Objective

- **Goal:**
  - Introduce you to JPA and Hibernate using a modern, hands-on Spring Boot approach.
  - Understand the evolution from JDBC to JPA/Hibernate, and how Spring Boot simplifies working with them.

2. What Will You Learn?

- **The World Before JPA:**
  - Understand how data access worked with JDBC and Spring JDBC.
- **Why JPA? Why Hibernate?**
  - Learn what problems JPA and Hibernate solve.
  - Understand the difference between JPA (the specification) and Hibernate (an implementation).
- **Spring Boot with JPA:**
  - See how Spring Boot and Spring Data JPA make JPA/Hibernate usage easy and powerful.
- **JPA Terminology:**
  - Concepts like Entity and Mapping.

3. Hands-On Learning Approach

You'll build a simple JPA application step-by-step:

1. **Create a Spring Boot Project**
   - Use H2 as an in-memory database (no external setup needed).
2. **Create a COURSE Table**
   - Set up your sample database schema in H2.
3. **Use Spring JDBC**
   - Interact with the COURSE table using traditional Spring JDBC.
4. **Use JPA and Hibernate**
   - Do the same operations with JPA and Hibernate to understand the improvements.
5. **Use Spring Data JPA**
   - Leverage the power and simplicity of Spring Data JPA for database operations.

4. Why This Sequence?

- **Starts with basics (JDBC) to show the challenges and boilerplate.**

- **Progresses to JPA and Hibernate to see how they abstract and simplify data access.**
- **Ends with Spring Data JPA, the modern standard for Spring Boot apps, which makes working with JPA even easier.**

5. Key Takeaways for This Section

- Clear understanding of **JPA and Hibernate** and their roles in Java persistence.
- Practical, step-by-step exposure to **building a data-driven app** using Spring Boot.
- See the **evolution from JDBC → JPA/Hibernate → Spring Data JPA.**
- Get comfortable with **JPA terminology** (like Entity and Mapping).
- Learn how **Spring Boot** brings all of this together with minimal configuration.

In summary:
This section is designed to give you a practical, hands-on introduction to JPA and Hibernate, while also showing how Spring Boot and Spring Data JPA make modern Java persistence simple and efficient. By the end, you'll have built a simple JPA app using best practices!

Creating a Spring Boot Project with Spring Initializr

1. Using Spring Initializr (start.spring.io)

- **Spring Initializr** is the easiest way to create a new Spring Boot project.
- **Steps:**
  - Go to start.spring.io.
  - Choose **Maven** as the build tool and **Java** as the language.
  - Select a suitable **Spring Boot version** (e.g., 3.0.0 M3 or any later stable release).
    - **Tip:** Avoid using "SNAPSHOT" versions as they are under development.
  - Set **Group ID** (e.g., com.in28minutes.springboot).
  - Set **Artifact ID** (e.g., learn-jpa-and-hibernate).
  - **Java Version:** Use Java 17 or newer (required for Spring Boot 3).

2. Adding Dependencies

- Add the following dependencies to your project:
  - **Spring Web:** For building web applications and REST APIs.
  - **Spring Data JDBC:** For working with databases using plain JDBC.
  - **Spring Data JPA:** For using JPA and Hibernate for ORM (Object Relational Mapping).
  - **H2 Database:** For an in-memory database, ideal for learning and testing.

3. Generating and Importing the Project

- Click **Generate** to download the zip file.
- **Extract** the zip file to your local machine (e.g., C:/).

- **Import the project into your IDE** (e.g., Eclipse):
  - Use File > Import > Maven > Existing Maven Projects.
  - Browse to the extracted folder and finish the import.
- Wait for the dependencies to download and the project to build (may take several minutes if it's your first time).

4. Project Structure

- After import, you should see:
  - src/main/java – Java source code
  - src/main/resources – configuration files
  - src/test/java – test code
  - pom.xml – Maven configuration
- The pom.xml should include:
  - spring-boot-starter-web
  - spring-boot-starter-data-jdbc
  - spring-boot-starter-data-jpa
  - h2database
  - spring-boot-starter-test (for testing)

5. Running the Application

- Locate the main class (e.g., LearnJpaAndHibernateApplication.java).
- **Right-click > Run As > Java Application** to start the app.
- **Console Output:**
  - You'll see logs indicating initialization of JDBC, JPA, Hibernate, and Tomcat (embedded server) starting on port 8080.
  - The startup is typically very fast (a couple of seconds).

6. Key Takeaways

- **Spring Initializr** makes it fast and easy to scaffold a new Spring Boot project with all the needed dependencies.
- **Spring Boot 3** requires Java 17 or higher.
- **Importing into Eclipse** (or your preferred IDE) sets up the project for immediate development.
- The application is ready to run with all the essential dependencies for web, JDBC, JPA, and an in-memory database.
- The Spring Boot magic (auto-configuration, embedded server, etc.) is visible in the startup logs—these will be explained in detail in upcoming steps.

In summary:
You learned how to quickly create, import, and launch a Spring Boot project for working with JPA and Hibernate, setting a strong foundation for further exploration and hands-on learning.

Connecting to H2 Database and Creating

the Course Table

1. Setting up H2 In-Memory Database in Spring Boot

- **H2 is an in-memory database** used for development and testing.
- When you start your Spring Boot app, H2 runs in memory and disappears when the app stops.

2. Enabling H2 Console for Easy Database Access

- To view and interact with your database, enable the H2 web console:
  - In application.properties, add:
    spring.h2.console.enabled=true
- **Access the console:**
  - After starting your app, visit:
    http://localhost:8080/h2-console

3. Configuring the Database URL

- By default, Spring Boot generates a dynamic H2 database URL each run, which can be inconvenient.
- **Set a static database URL in application.properties:**
  spring.datasource.url=jdbc:h2:mem:testdb
  - This ensures the same database name (testdb) is used every run.

4. Connecting to the H2 Console

- Use the **JDBC URL:**
  jdbc:h2:mem:testdb
- The **username** is usually sa (default), and the password field can be left blank (unless you configure one).

5. Creating the course Table with schema.sql

- **Create a file named schema.sql** in src/main/resources.
- Put your table creation SQL inside:

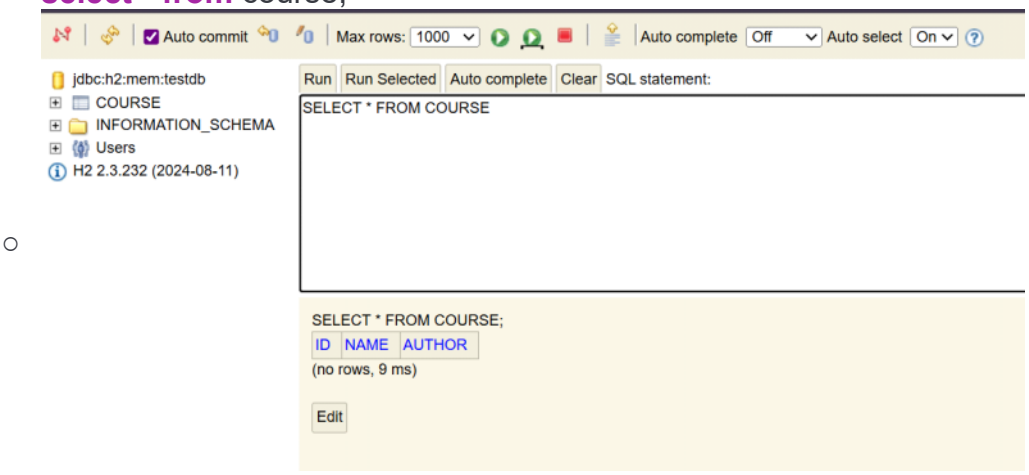
```
create table course
(
  id bigint not null,
  name varchar(255) not null,
  author varchar(255) not null,
  primary key (id)
);
```

- **Spring Boot will automatically run schema.sql on startup** to create the table in H2.

6. Verifying the Table

- After app startup:
  - Go to the H2 console (/h2-console), connect, and run:

select * from course;



- You should see the table structure with columns: id, name, and author.

- **If the table does not appear:**
  - Ensure schema.sql is in src/main/resources.
  - Check the file name and SQL syntax for typos.
  - Restart the app after making changes.

7. Summary Table: Key Configuration

File	Key Configuration/Code
application.properties	spring.application.name=learn-jpa-and-hibernate
	spring.h2.console.enabled=true
	spring.datasource.url=jdbc:h2:mem:testdb
schema.sql	create table course (id bigint not null, name varchar(255) not null, author varchar(255) not null, primary key (id));

8. Next Steps

- With the table ready, you can now practice using **Spring JDBC, JPA, Hibernate, and Spring Data JPA** to interact with the course table.

In summary:
You've set up an in-memory H2 database, enabled its web console, configured a fixed database URL, and created your course table automatically using schema.sql. This foundation allows you to start experimenting with different Spring data access technologies in the next steps!

application.properties
```
spring.application.name=learn-jpa-and-hibernate
spring.h2.console.enabled=true
spring.datasource.url=jdbc:h2:mem:testdb
```

schema.sql
```
create table course
(
id bigint not null,
name varchar(255) not null,
```

```
author varchar(255) not null,
primary key (id)

);
```

Spring JDBC: Overview and Basic Data Operations

1. Recap: Current Setup

- **H2 in-memory database** is running with a course table.
- You can access and modify the data using the H2 Console (web interface).

2. Performing SQL Operations Directly in H2 Console

- **Insert Data:**
  ```
  insert into course (id, name, author)
  values (1, 'Learn AWS', 'in28minutes');
  ```
- **View Data:**
  ```
  select * from course;
  ```
- **Delete Data:**
  ```
  delete from course where id=1;
  ```
- **Tip:**
  - Use "Run Selected" in the H2 Console to execute only the highlighted query.
  - Data in the in-memory H2 database is **lost every time you restart the application.**

3. What is SQL?

- **SQL (Structured Query Language):**
  - The language used to interact with relational databases (insert, select, delete, update, etc.).

4. JDBC and Spring JDBC: What's the Difference?

- **JDBC (Java Database Connectivity):**
  - Standard Java API for database access.
  - Requires lots of boilerplate code (manage connections, statements, exceptions, etc.).
- **Spring JDBC:**
  - A Spring Framework module that simplifies JDBC operations.
  - You **still write SQL queries**, but **write much less Java code.**

- Handles much of the repetitive work (resource management, error handling, etc.).

Example Comparison

- **Plain JDBC:** Many lines of code for a simple query (open connection, prepare statement, execute, handle exceptions, close connection).

```
public void deleteTodo(int id) {
    PreparedStatement st = null;
    try {
        st = db.conn.prepareStatement("delete from todo where id=?");
        st.setInt(1, id);
        st.execute();
    } catch (SQLException e) {
        logger.fatal("Query Failed : ", e);
    } finally {
        if (st != null) {
            try {st.close();}
            catch (SQLException e) {}
        }
    }
}
```

- **Spring JDBC:** Only a few lines—just specify the query and parameters; Spring does the rest.

```
public void deleteTodo(int id) {
    jdbcTemplate.update("delete from todo where id=?", id);
}
```

5. Workflow for Using Spring JDBC

- **Define your SQL queries** (same as you would in the H2 console).
- **Use Spring JDBC classes** (like JdbcTemplate) to execute those queries from Java code.
- **Benefit:**
  - Focus on your logic and queries, not on repetitive boilerplate.

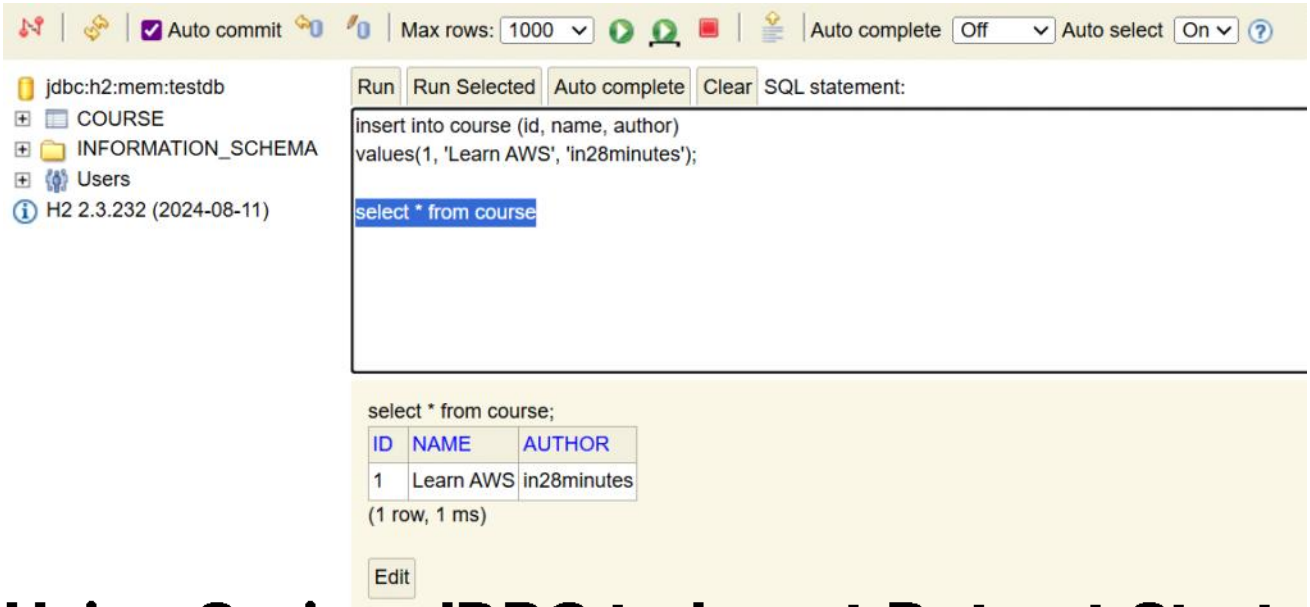
6. Practical Advice

- **Keep your frequently-used queries in a text file (e.g., notes.txt)** for easy reuse, since H2 is in-memory and resets on restart.

7. Key Takeaways

- **Spring JDBC lets you execute SQL queries with minimal Java code.**
- You still need to know SQL, but you avoid the repetitive setup and teardown code required by plain JDBC.
- **Common operations** (insert, select, delete) are the same in SQL and when using Spring JDBC—only the Java integration is easier.

In summary:
You learned how to manually insert, view, and delete data in the H2 database using SQL. The next step is to use **Spring JDBC** to execute the same queries from your Java code, letting you interact with your database programmatically in a much simpler way than plain JDBC.



Using Spring JDBC to Insert Data at Startup in Spring Boot

1. Goal

- **Automate data insertion** into the course table at application startup using Spring JDBC.

2. Key Components

A. CourseJdbcRepository

- **Purpose:** Encapsulates database operations for the course table.
- **Uses:**
  - **JdbcTemplate:** A Spring utility that simplifies JDBC operations.
  - **@Repository** annotation: Indicates this class interacts with the database and enables component scanning.
- **Code:**

```
@Repository
public class CourseJdbcRepository {
    @Autowired
    private JdbcTemplate springJdbcTemplate;

    private static String INSERT_QUERY =
        """
        insert into course (id, name, author)
        values(1, 'Learn AWS', 'in28minutes');
        """;

    public void insert() {
        springJdbcTemplate.update(INSERT_QUERY);
    }
}
```

- **Text Blocks (""""):**
  - Java feature for multi-line strings, making SQL queries easier to read and maintain.

B. CourseJdbcCommandLineRunner

- **Purpose:** Executes logic at Spring Boot application startup.
- **Implements:**
  - **CommandLineRunner:** Interface with run() method that executes after the application context is loaded.
- **Uses:**
  - **@Component:** Registers the class as a Spring bean.
  - **@Autowired:** Injects the repository.
- **Code:**

```
@Component
public class CourseJdbcCommandLineRunner implements CommandLineRunner {
    @Autowired
    private CourseJdbcRepository repository;

    @Override
    public void run(String... args) throws Exception {
        repository.insert();
    }
}
```

3. How It Works

- **At startup:**
  1. **Spring Boot creates all beans** (using component scanning).
  2. **CommandLineRunner beans** have their run() method executed.
  3. CourseJdbcCommandLineRunner.run() calls repository.insert().
  4. CourseJdbcRepository.insert() executes the SQL insert statement using JdbcTemplate.
  5. The new row is added to the course table in H2.

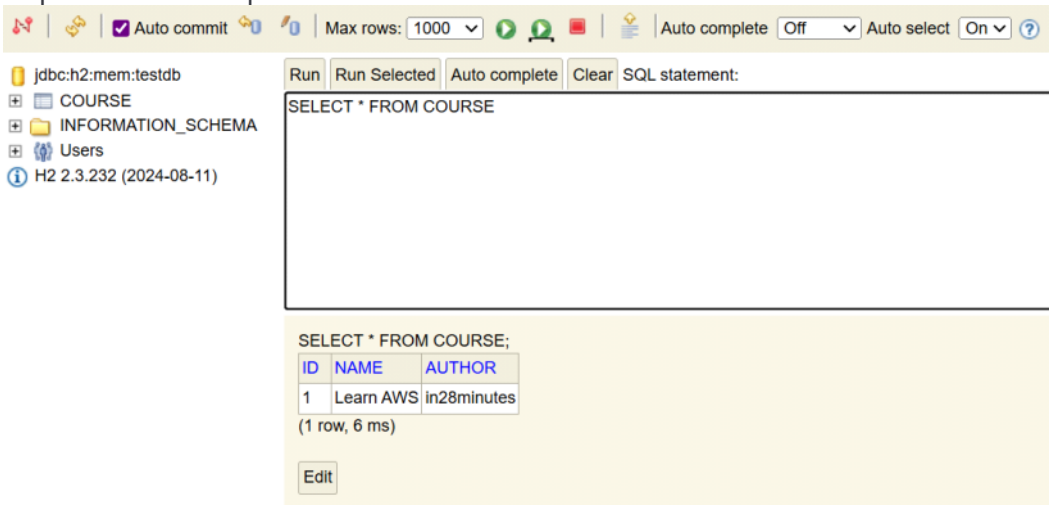
4. Verifying the Result

- **Open the H2 Console** (http://localhost:8080/h2-console).
- **Run:**

```
select * from course;
```

- **You should see:**

```
id | name          | author
1  | Learn AWS     | in28minutes
```



5. Best Practices & Notes

- **@Repository** is used for data access classes.
- **@Component** makes a class eligible for component scanning and auto-detection.
- **@Autowired** handles dependency injection automatically.
- **CommandLineRunner** is great for executing setup or test code at application startup.
- **JdbcTemplate** dramatically reduces JDBC boilerplate.

6. Key Takeaways

- **Spring JDBC with JdbcTemplate** allows you to execute SQL with minimal code.
- **Startup logic** (like data seeding) can be automated using CommandLineRunner.
- **Code is clean and focused:** Business logic in repositories, startup logic in runners.

In summary:
You learned how to use Spring JDBC (JdbcTemplate) to insert data into the database automatically at application startup in a Spring Boot application, using best practices for structure and dependency injection.

CourseJdbcRepository.java

```
package com.in28minutes.springboot.learn_jpa_and_hibernate.course.jdbc;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Repository;

@Repository
public class CourseJdbcRepository {

	@Autowired
	private JdbcTemplate springJdbcTemplate;

	private static String INSERT_QUERY =
			"""
				insert into course (id, name, author)
				values(1, 'Learn AWS', 'in28minutes');

			""";

	public void insert() {
		springJdbcTemplate.update(INSERT_QUERY);
	}

}
```

CourseJdbcCommandLineRunner.java

```
package com.in28minutes.springboot.learn_jpa_and_hibernate.course.jdbc;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

@Component
public class CourseJdbcCommandLineRunner implements CommandLineRunner {

	@Autowired
	private CourseJdbcRepository repository;
```

```
	@Override
	public void run(String... args) throws Exception {
		repository.insert();
	}

}
```

Spring JDBC: Inserting and Deleting Data Using a Repository Pattern

1. CourseJdbcRepository: Encapsulating Data Operations

- **Purpose:**
  Centralizes all JDBC operations related to the course table.
- **Key Features:**
  - Uses **JdbcTemplate** (injected via @Autowired) for executing SQL statements.
  - Stores SQL queries as static string constants, using **parameter placeholders (?)** for safer, dynamic values.

Code Highlights:

```
@Repository
public class CourseJdbcRepository {

   @Autowired
   private JdbcTemplate springJdbcTemplate;

   private static String INSERT_QUERY = """
      insert into course (id, name, author)
      values(?, ?, ?);
   """;

   private static String DELETE_QUERY = """
      delete from course
      where id=?
   """;

   public void insert(Course course) {
      springJdbcTemplate.update(INSERT_QUERY,
```

```
            course.getId(), course.getName(), course.getAuthor());
    }

    public void deleteById(long id) {
        springJdbcTemplate.update(DELETE_QUERY, id);
    }
}
```

- **Benefits of parameterized queries:**
  - Prevents SQL injection.
  - Cleanly separates SQL from values.

2. CourseJdbcCommandLineRunner: Running Data Logic at Startup

- **Purpose:**
  Automates inserting and deleting course records as soon as the Spring Boot app starts.
- **Implements:**
  CommandLineRunner interface, which runs its run() method after the application context loads.

Code Highlights:

```
@Component
public class CourseJdbcCommandLineRunner implements CommandLineRunner {

    @Autowired
    private CourseJdbcRepository repository;

    @Override
    public void run(String... args) throws Exception {
        repository.insert(new Course(1, "Learn AWS Now!", "in28minutes"));
        repository.insert(new Course(2, "Learn Azure Now!", "in28minutes"));
        repository.insert(new Course(3, "Learn DevOps Now!", "in28minutes"));

        repository.deleteById(1);
    }
}
```

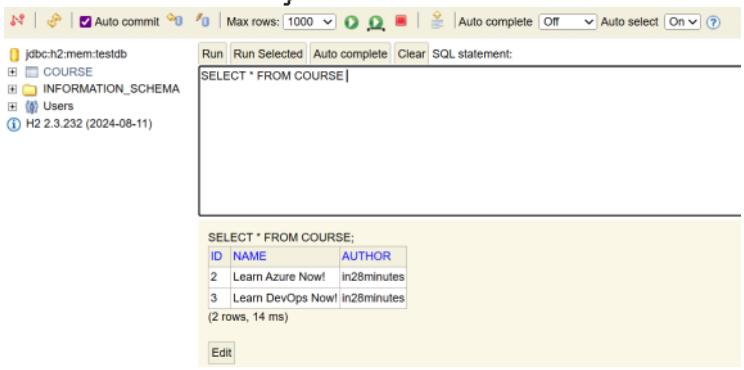
- **What happens at startup:**
  - Three courses are inserted into the database.
  - The course with id=1 is immediately deleted.
  - The final state of the database will have only courses with id=2 and id=3.

3. How Spring JDBC Works Here

- **Inserting:**
  insert(Course course) uses the INSERT_QUERY with values from the provided Course object.
- **Deleting:**
  deleteById(long id) uses the DELETE_QUERY with the provided id.

4. Verifying the Results

- **H2 Console:**
  - Go to http://localhost:8080/h2-console and connect.
  - Run: SELECT * FROM course;
  - You should see only the courses with id=2 and id=3.



5. Best Practices Illustrated

- **Repository Pattern:**
  Keeps data access logic separate and reusable.
- **Parameterization:**
  Always use ? in SQL queries for values, not string concatenation.
- **Dependency Injection:**
  @Autowired injects dependencies cleanly.
- **Startup Data Seeding:**
  Use CommandLineRunner for demo/testing data.

6. Key Takeaways

- **Spring JDBC with JdbcTemplate** allows you to execute SQL with minimal, readable code.
- **Repository classes** are the right place for encapsulating and organizing database operations.
- **CommandLineRunner** is useful for running initial setup, demo, or testing logic on application start.

In summary:
You now have a simple, maintainable way to perform basic data operations (insert and delete) on the course table using Spring JDBC in a Spring Boot application. This sets the foundation for further data access patterns and for exploring more advanced features like JPA and Spring Data JPA.

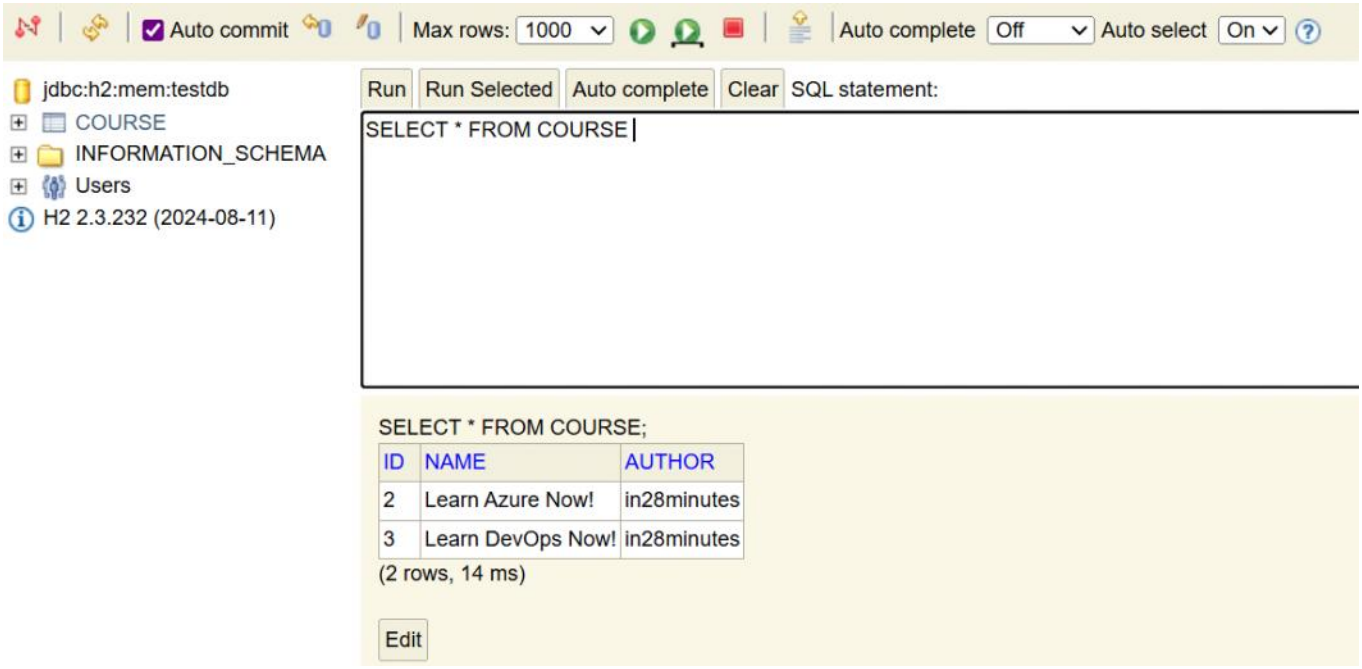
CourseJdbcRepository.java

```
package com.in28minutes.springboot.learn_jpa_and_hibernate.course.jdbc;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Repository;

import com.in28minutes.springboot.learn_jpa_and_hibernate.course.Course;

@Repository
public class CourseJdbcRepository {
```

```
	@Autowired
	private JdbcTemplate springJdbcTemplate;

	private static String INSERT_QUERY =

			"""
				insert into course (id, name, author)
				values(?, ?, ?);

			""";

	private static String DELETE_QUERY =

			"""
				delete from course
				where id=?

			""";

	public void insert(Course course) {
		springJdbcTemplate.update(INSERT_QUERY,
				course.getId(), course.getName(), course.getAuthor());
	}

	public void deleteById(long id) {
		springJdbcTemplate.update(DELETE_QUERY, id);
	}
}
```

CourseJdbcCommandLineRunner.java

```
package com.in28minutes.springboot.learn_jpa_and_hibernate.course.jdbc;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

import com.in28minutes.springboot.learn_jpa_and_hibernate.course.Course;

@Component
public class CourseJdbcCommandLineRunner implements CommandLineRunner {

	@Autowired
	private CourseJdbcRepository repository;

	@Override
	public void run(String... args) throws Exception {
		repository.insert(new Course(1, "Learn AWS Now!", "in28minutes"));
		repository.insert(new Course(2, "Learn Azure Now!", "in28minutes"));
		repository.insert(new Course(3, "Learn DevOps Now!", "in28minutes"));

		repository.deleteById(1);

	}

}
```



Spring JDBC: Retrieving Data and Mapping Results to Beans

1. Goal

- **Retrieve data from the database** (specifically, by course ID) using Spring JDBC.
- **Map the result of a SQL query** to a Java object (Course).

2. Writing the Query and Repository Method

- **SQL SELECT Query:**
  `select * from course where id=?`
- **Repository Method:**
  - Method: `public Course findById(long id)`
  - Uses `JdbcTemplate.queryForObject()` to:
    - Execute the query.
    - Map the result to a Course object.
  - Uses `BeanPropertyRowMapper<>(Course.class)` for automatic mapping based on property names.

Code:

```
public Course findById(long id) {
   // ResultSet -> Bean => Row Mapper =>
   return springJdbcTemplate.queryForObject(
      SELECT_QUERY,
      new BeanPropertyRowMapper<>(Course.class),
      id);
```

```
}
```

3. Row Mapping with BeanPropertyRowMapper

- **What is a RowMapper?**
  - Converts a row from the SQL result set into a Java object.
- **BeanPropertyRowMapper:**
  - Maps columns to Java bean properties by matching names.
  - Works when **database column names and bean property names are identical** (case-insensitive).
  - Requires **getters and setters** in the Java class.

4. Course Entity Class

- **Fields:** id, name, author
- **Constructors:** Default and parameterized
- **Getters and Setters:** Required for BeanPropertyRowMapper to populate the object.
- **toString() Method:** For readable output when printing the object.

Code:

```
public class Course {
    private long id;
    private String name;
    private String author;

    // Default constructor
    public Course() {}

    // Parameterized constructor
    public Course(long id, String name, String author) { ... }

    // Getters and setters for all fields

    @Override
    public String toString() { ... }
}
```

5. CommandLineRunner: Testing the Flow

- **Startup Logic:**
  - Insert three courses.
  - Delete the course with id=1.
  - Retrieve and print courses with id=2 and id=3.

Code:

```
@Override
public void run(String... args) throws Exception {
    repository.insert(new Course(1, "Learn AWS Now!", "in28minutes"));
    repository.insert(new Course(2, "Learn Azure Now!", "in28minutes"));
    repository.insert(new Course(3, "Learn DevOps Now!", "in28minutes"));

    repository.deleteById(1);
```

```
    System.out.println(repository.findById(2));
    System.out.println(repository.findById(3));
}
```

- **Expected output:**
  Course [id=2, name=Learn Azure Now!, author=in28minutes]
  Course [id=3, name=Learn DevOps Now!, author=in28minutes]

6. Common Pitfall: Missing Setters

- **Issue:** If your Java bean (Course) has only getters and no setters, **BeanPropertyRowMapper cannot populate fields** from the result set, leading to empty or default values.
- **Solution:**
  - Add setters for all bean properties.

7. Key Takeaways

- **Spring JDBC can retrieve and map SQL results directly to Java objects** using row mappers.
- **BeanPropertyRowMapper** is convenient when your table and bean property names match.
- **Getters and setters are essential** for mapping to work.
- **CommandLineRunner** is a useful tool for quick data operations/testing at startup.

In summary:
You learned how to use Spring JDBC to retrieve a specific row from the database and map it to a Java object, and why having proper getters and setters is crucial for this automatic mapping to succeed.

CourseJdbcRepository.java

```
package com.in28minutes.springboot.learn_jpa_and_hibernate.course.jdbc;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.BeanPropertyRowMapper;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Repository;

import com.in28minutes.springboot.learn_jpa_and_hibernate.course.Course;

@Repository
public class {

	@Autowired
	private JdbcTemplate springJdbcTemplate;

	private static String INSERT_QUERY =

			"""
				insert into course (id, name, author)
				values(?, ?, ?);

			""";

	private static String DELETE_QUERY =
```

```
                """
                        delete from course
                        where id=?

                """;

        private static String SELECT_QUERY =

                """
                        select * from course
                        where id=?

                """;

        public void insert(Course course) {
                springJdbcTemplate.update(INSERT_QUERY,
                        course.getId(), course.getName(), course.getAuthor());
        }

        public void deleteById(long id) {
                springJdbcTemplate.update(DELETE_QUERY, id);
        }

        public Course findById(long id) {
                // ResultSet -> Bean => Row Mapper =>
                return springJdbcTemplate.queryForObject(SELECT_QUERY,
                        new BeanPropertyRowMapper<>(Course.class), id);
        }
}
```

CourseJdbcCommandLineRunner.java

```
package com.in28minutes.springboot.learn_jpa_and_hibernate.course.jdbc;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

import com.in28minutes.springboot.learn_jpa_and_hibernate.course.Course;

@Component
public class CourseJdbcCommandLineRunner implements CommandLineRunner {

        @Autowired
        private CourseJdbcRepository repository;

        @Override
        public void run(String... args) throws Exception {
                repository.insert(new Course(1, "Learn AWS Now!", "in28minutes"));
                repository.insert(new Course(2, "Learn Azure Now!", "in28minutes"));
                repository.insert(new Course(3, "Learn DevOps Now!", "in28minutes"));

                repository.deleteById(1);

                System.out.println(repository.findById(2));
                System.out.println(repository.findById(3));

        }

}
```

Course.java

```
package com.in28minutes.springboot.learn_jpa_and_hibernate.course;

public class {
	private long id;
	private String name;
	private String author;

	public Course() {

	}

	public Course(long id, String name, String author) {
		super();
		this.id = id;
		this.name = name;
		this.author = author;
	}

	public long getId() {
		return id;
	}

	public String getName() {
		return name;
	}

	public String getAuthor() {
		return author;
	}

	public void setId(long id) {
		this.id = id;
	}

	public void setName(String name) {
		this.name = name;
	}

	public void setAuthor(String author) {
		this.author = author;
	}

	@Override
	public String toString() {
		return "Course [id=" + id + ", name=" + name + ", author=" + author + "]";
	}

	// Constructor
	// getters
	// toString

}
```

```
2025-07-27T03:04:59.636+05:30  INFO 18824 --- [learn-jpa-and-hibernate] [
2025-07-27T03:04:59.802+05:30  WARN 18824 --- [learn-jpa-and-hibernate] [
2025-07-27T03:05:00.460+05:30  INFO 18824 --- [learn-jpa-and-hibernate] [
2025-07-27T03:05:00.562+05:30  INFO 18824 --- [learn-jpa-and-hibernate] [
2025-07-27T03:05:00.575+05:30  INFO 18824 --- [learn-jpa-and-hibernate] [
Course [id=2, name=Learn Azure Now!, author=in28minutes]
Course [id=3, name=Learn DevOps Now!, author=in28minutes]
```

Spring Boot + JPA: CRUD Operations with EntityManager

1. application.properties Configuration

application.properties

```
spring.application.name=learn-jpa-and-hibernate
spring.h2.console.enabled=true
spring.datasource.url=jdbc:h2:mem:testdb
spring.jpa.show-sql=true
```

- Enables the H2 in-memory database and its web console for easy access.
- Configures the datasource URL.
- spring.jpa.show-sql=true prints the SQL generated by Hibernate/JPA to the console, helping to understand what happens behind the scenes.

2. Course Entity Definition

Course.java

```java
package com.in28minutes.springboot.learn_jpa_and_hibernate.course;

import jakarta.persistence.Entity;
import jakarta.persistence.Id;

@Entity
public class Course {

	@Id
	private long id;

	private String name;

	private String author;

	public Course() {

	}

	public Course(long id, String name, String author) {
		super();
		this.id = id;
		this.name = name;
		this.author = author;
	}
```

```
	public long getId() {
		return id;
	}

	public String getName() {
		return name;
	}

	public String getAuthor() {
		return author;
	}

	public void setId(long id) {
		this.id = id;
	}

	public void setName(String name) {
		this.name = name;
	}

	public void setAuthor(String author) {
		this.author = author;
	}

	@Override
	public String toString() {
		return "Course [id=" + id + ", name=" + name + ", author=" + author + "]";
	}

	// Constructor
	// getters
	// toString

}
```

- The class is annotated with @Entity to mark it as a JPA entity.
- The id field is annotated with @Id as the primary key.
- Contains default and parameterized constructors, getters, setters, and a toString() method for display.

3. JPA Repository Implementation Using EntityManager

CourseJpaRepository.java

```
package com.in28minutes.springboot.learn_jpa_and_hibernate.course.jpa;

import org.springframework.stereotype.Repository;

import com.in28minutes.springboot.learn_jpa_and_hibernate.course.Course;

import jakarta.persistence.EntityManager;
import jakarta.persistence.PersistenceContext;
import jakarta.transaction.Transactional;

@Repository
@Transactional
public class CourseJpaRepository {
```

```
	@PersistenceContext
	private EntityManager entityManager;

	public void insert(Course course) {
		entityManager.merge(course);
	}

	public Course findById(long id) {
		return entityManager.find(Course.class, id);
	}

	public void deleteById(long id) {
		Course course = entityManager.find(Course.class, id);
		entityManager.remove(course);
	}

}
```

- Annotated with @Repository for Spring to detect it as a repository bean.
- Annotated with @Transactional to ensure methods run in a transaction.
- EntityManager is injected using @PersistenceContext.
- merge() is used for both insert and update.
- find() retrieves an entity by its primary key.
- remove() deletes the entity.

4. CommandLineRunner to Execute Operations at Startup

CourseCommandLineRunner.java

```
package com.in28minutes.springboot.learn_jpa_and_hibernate.course;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

import com.in28minutes.springboot.learn_jpa_and_hibernate.course.jpa.CourseJpaRepository;

@Component
public class CourseCommandLineRunner implements CommandLineRunner {

//	@Autowired
//	private CourseJdbcRepository repository;

	@Autowired
	private CourseJpaRepository repository;

	@Override
	public void run(String... args) throws Exception {
		repository.insert(new Course(1, "Learn AWS Jpa!", "in28minutes"));
		repository.insert(new Course(2, "Learn Azure Jpa!", "in28minutes"));
		repository.insert(new Course(3, "Learn DevOps Jpa!", "in28minutes"));

		repository.deleteById(1);

		System.out.println(repository.findById(2));
		System.out.println(repository.findById(3));
```

```
    }
}
```

- Implements CommandLineRunner to run at application startup.
- Inserts three courses, deletes the one with id=1, and prints the remaining two by fetching them with their IDs.

5. Console Output and SQL Logging

When the application runs, Hibernate prints the SQL it executes:

```
Hibernate: insert into course (author,name,id) values (?,?,?)
Hibernate: insert into course (author,name,id) values (?,?,?)
Hibernate: insert into course (author,name,id) values (?,?,?)
Hibernate: delete from course where id=?
Hibernate: select c1_0.id,c1_0.author,c1_0.name from course c1_0 where c1_0.id=?
Hibernate: select c1_0.id,c1_0.author,c1_0.name from course c1_0 where c1_0.id=?
```

Printed output:

```
Course [id=2, name=Learn Azure Jpa!, author=in28minutes]
Course [id=3, name=Learn DevOps Jpa!, author=in28minutes]
```

```
2025-07-27T03:38:40.662+05:30  INFO 31036 --- [learn-jpa-and-hibernate] [           m
Hibernate: select c1_0.id,c1_0.author,c1_0.name from course c1_0 where c1_0.id=?
Hibernate: insert into course (author,name,id) values (?,?,?)
Hibernate: select c1_0.id,c1_0.author,c1_0.name from course c1_0 where c1_0.id=?
Hibernate: insert into course (author,name,id) values (?,?,?)
Hibernate: select c1_0.id,c1_0.author,c1_0.name from course c1_0 where c1_0.id=?
Hibernate: insert into course (author,name,id) values (?,?,?)
Hibernate: select c1_0.id,c1_0.author,c1_0.name from course c1_0 where c1_0.id=?
Hibernate: delete from course where id=?
Hibernate: select c1_0.id,c1_0.author,c1_0.name from course c1_0 where c1_0.id=?
Course [id=2, name=Learn Azure Jpa!, author=in28minutes]
Hibernate: select c1_0.id,c1_0.author,c1_0.name from course c1_0 where c1_0.id=?
Course [id=3, name=Learn DevOps Jpa!, author=in28minutes]
```

- Confirms that the operations are happening as expected and shows the underlying SQL.

Summary of the Flow

1. Spring Boot is configured to use H2 and log SQL.
2. The Course entity is mapped to a database table.
3. A custom repository uses JPA's EntityManager for CRUD operations.
4. Operations are executed at startup using CommandLineRunner.
5. Hibernate SQL and results are visible in the console for verification.

This approach demonstrates how to use JPA with Spring Boot for basic CRUD without writing SQL manually, leveraging automatic mapping and transaction management.

Transition from Spring JDBC to JPA

1. Comparison: JDBC, Spring JDBC, and JPA

- **JDBC**: Requires writing a lot of SQL queries and boilerplate Java code for every database operation.
- **Spring JDBC**: Reduces Java code but still requires you to write SQL queries for all operations.
- **JPA**: Removes the need to write SQL queries for basic CRUD operations. You focus on

mapping Java classes (entities) to database tables.

2. Mapping Entities to Tables

- Using JPA, entities (Java classes) are mapped directly to database tables.
- Example: The Course entity is mapped to the course table.

```
@Entity
public class Course {
    @Id
    private long id;
    private String name;
    private String author;
    // Constructors, getters, setters, toString...
}
```

- With the entity defined, you can use JPA to insert, find, and delete data without writing SQL.

3. Using EntityManager for CRUD Operations

- **EntityManager** is the central JPA API to manage entities.
- Typical operations:
  - merge(course): Insert or update an entity.
  - find(Course.class, id): Retrieve an entity by its primary key.
  - remove(course): Delete an entity.

```
@PersistenceContext
private EntityManager entityManager;

public void insert(Course course) {
    entityManager.merge(course);
}

public Course findById(long id) {
    return entityManager.find(Course.class, id);
}

public void deleteById(long id) {
    Course course = entityManager.find(Course.class, id);
    entityManager.remove(course);
}
```

- No need to write SQL queries for these operations; JPA handles it based on your entity mappings.

4. Reviewing the Queries Executed by JPA

- Although you don't write SQL in your code, JPA/Hibernate still generates and executes SQL queries in the background.
- With spring.jpa.show-sql=true, you can see these queries in your console/logs.

Example log output:

```
Hibernate: insert into course (author,name,id) values (?,?,?)
Hibernate: delete from course where id=?
```

Hibernate: select c1_0.id,c1_0.author,c1_0.name from course c1_0 where c1_0.id=?

- For insert, Hibernate generates an INSERT statement.
- For deleteById, Hibernate first performs a SELECT to check for existence, then executes a DELETE.
- For findById, Hibernate performs a SELECT with the given id.

5. Code Sequence in Practice

```
// Insert three courses
repository.insert(new Course(1, "Learn AWS Jpa!", "in28minutes"));
repository.insert(new Course(2, "Learn Azure Jpa!", "in28minutes"));
repository.insert(new Course(3, "Learn DevOps Jpa!", "in28minutes"));

// Delete course with id=1
repository.deleteById(1);

// Retrieve and print remaining courses
System.out.println(repository.findById(2));
System.out.println(repository.findById(3));
```

Corresponding SQL (printed by Hibernate):

```
Hibernate: insert into course (author,name,id) values (?,?,?)
Hibernate: insert into course (author,name,id) values (?,?,?)
Hibernate: insert into course (author,name,id) values (?,?,?)
Hibernate: select c1_0.id,c1_0.author,c1_0.name from course c1_0 where c1_0.id=?
Hibernate: delete from course where id=?
Hibernate: select c1_0.id,c1_0.author,c1_0.name from course c1_0 where c1_0.id=?
Hibernate: select c1_0.id,c1_0.author,c1_0.name from course c1_0 where c1_0.id=?
```

Printed Output:

```
Course [id=2, name=Learn Azure Jpa!, author=in28minutes]
Course [id=3, name=Learn DevOps Jpa!, author=in28minutes]
```

```
Hibernate: select c1_0.id,c1_0.author,c1_0.name from course c1_0 where c1_0.id=?
Hibernate: insert into course (author,name,id) values (?,?,?)
Hibernate: select c1_0.id,c1_0.author,c1_0.name from course c1_0 where c1_0.id=?
Hibernate: insert into course (author,name,id) values (?,?,?)
Hibernate: select c1_0.id,c1_0.author,c1_0.name from course c1_0 where c1_0.id=?
Hibernate: insert into course (author,name,id) values (?,?,?)
Hibernate: select c1_0.id,c1_0.author,c1_0.name from course c1_0 where c1_0.id=?
Hibernate: delete from course where id=?
Hibernate: select c1_0.id,c1_0.author,c1_0.name from course c1_0 where c1_0.id=?
Course [id=2, name=Learn Azure Jpa!, author=in28minutes]
Hibernate: select c1_0.id,c1_0.author,c1_0.name from course c1_0 where c1_0.id=?
Course [id=3, name=Learn DevOps Jpa!, author=in28minutes]
```

6. Key Takeaways

- JPA significantly reduces boilerplate by removing the need for most SQL in your code.
- EntityManager provides a clean API for CRUD operations.
- JPA still generates SQL under the hood and executes it on your behalf.
- You can always review the actual SQL via logging, which helps with understanding and

debugging.

7. Looking Ahead

- While JPA with EntityManager is much simpler than JDBC, Spring Data JPA offers even more convenience.
- Spring Data JPA reduces the code further, providing ready-to-use repository interfaces for common patterns.
- The next step explores why and how to use Spring Data JPA for even greater productivity.

JPA Example

```
@Repository
public class PersonJpaRepository {

  @PersistenceContext
  EntityManager entityManager;

  public Person findById(int id) {
    return entityManager.find(Person.class, id);
  }

  public Person update(Person person) {
    return entityManager.merge(person);
  }

  public Person insert(Person person) {
    return entityManager.merge(person);
  }

  public void deleteById(int id) {........
```

Spring Data JPA Example

```
public interface TodoRepository extends JpaRepository<Todo, Integer>{
```

Spring Data JPA: Making JPA Even Simpler

1. Recap: Evolution from JDBC to Spring Data JPA

Approach	SQL Required	Java Code Required	Abstraction Level
JDBC	High	High	Low
Spring JDBC	High	Lower	Moderate
JPA	Low	Low	High (EntityManager)
Spring Data JPA	None	Very Low	Very High (Repository)

- **JDBC:** Write all SQL and lots of Java code.
- **Spring JDBC:** Still write SQL, but less Java code.
- **JPA:** No SQL needed for most operations; map entities and use EntityManager.
- **Spring Data JPA:** Even simpler—no need to use EntityManager directly. Just define a repository interface and use built-in methods.

2. Defining the Spring Data JPA Repository

- No need to write implementation code.
- Create an interface that extends JpaRepository<Entity, IDType>.

```
package com.in28minutes.springboot.learn_jpa_and_hibernate.course.springdatajpa;

import org.springframework.data.jpa.repository.JpaRepository;

import com.in28minutes.springboot.learn_jpa_and_hibernate.course.Course;

public interface CourseSpringDataJpaRepository extends JpaRepository<Course, Long>{

}
```

- **Explanation:**
  - JpaRepository is a generic interface provided by Spring Data JPA.
  - By extending it, you get a complete set of CRUD and query methods automatically.
  - You specify the entity type (Course) and the type of its primary key (Long).

3. Using the Repository in Your Application

- Inject the repository and call its methods. No need to use EntityManager or write any SQL.

```
package com.in28minutes.springboot.learn_jpa_and_hibernate.course;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

import
com.in28minutes.springboot.learn_jpa_and_hibernate.course.springdatajpa.CourseSpringDat
aJpaRepository;

@Component
public class CourseCommandLineRunner implements CommandLineRunner {

//	@Autowired
//	private CourseJdbcRepository repository;

//	@Autowired
//	private CourseJpaRepository repository;

	@Autowired
	private CourseSpringDataJpaRepository repository;

	@Override
	public void run(String... args) throws Exception {
		repository.save(new Course(1, "Learn AWS Jpa!", "in28minutes"));
		repository.save(new Course(2, "Learn Azure Jpa!", "in28minutes"));
		repository.save(new Course(3, "Learn DevOps Jpa!", "in28minutes"));

		repository.deleteById(1l);

		System.out.println(repository.findById(2l));
		System.out.println(repository.findById(3l));
```

```
        }

}
```

- **Explanation:**
  - .save(entity): Inserts or updates an entity.
  - .deleteById(id): Deletes an entity by its primary key.
  - .findById(id): Retrieves an entity by its primary key (returns Optional<Course>).
  - All these methods are inherited from JpaRepository—no implementation needed.

4. Advantages of Spring Data JPA

- No need to write SQL for standard CRUD operations.
- No need to use or inject EntityManager.
- Repository interface exposes a rich set of operations (save, delete, find, count, exists, findAll, etc.).
- Cleaner, more maintainable codebase.

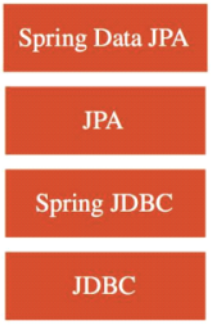
5. Common Repository Methods Provided

- save(entity) — Insert or update
- findById(id) — Find one entity by primary key
- findAll() — Get all entities
- deleteById(id) — Delete by primary key
- count() — Get the number of entities
- existsById(id) — Check if an entity exists
- And many more...

6. Summary Table from Slides

JDBC to Spring JDBC to JPA to Spring Data JPA

- **JDBC**
  - Write a lot of SQL queries! (*delete from todo where id=?*)
  - And write a lot of Java code
- **Spring JDBC**
  - Write a lot of SQL queries (*delete from todo where id=?*)
  - BUT lesser Java code
- **JPA**
  - Do NOT worry about queries
  - Just Map Entities to Tables!
- **Spring Data JPA**
  - Let's make JPA even more simple!
  - I will take care of everything!



Approach	SQL Queries	Java Code	Abstraction
JDBC	Lots	Lots	Manual
Spring JDBC	Lots	Less	Template
JPA	Little/None	Less	EntityManager

Spring Data JPA	None	Very Little	Repository

7. Key Takeaway

- With **Spring Data JPA**, you only define your entity and a repository interface.
- All CRUD operations and many more are available out-of-the-box—no SQL, no implementation code, no EntityManager.
- It's the simplest and most productive way to interact with a relational database in a Spring Boot application.

CourseSpringDataJpaRepository.java

```
package com.in28minutes.springboot.learn_jpa_and_hibernate.course.springdatajpa;

import org.springframework.data.jpa.repository.JpaRepository;

import com.in28minutes.springboot.learn_jpa_and_hibernate.course.Course;

public interface CourseSpringDataJpaRepository extends JpaRepository<Course, Long>{

}
```

CourseCommandLineRunner.java

```
package com.in28minutes.springboot.learn_jpa_and_hibernate.course;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

import
com.in28minutes.springboot.learn_jpa_and_hibernate.course.springdatajpa.CourseSpringDataJpaR
epository;

@Component
public class CourseCommandLineRunner implements CommandLineRunner {

//	@Autowired
//	private CourseJdbcRepository repository;

//	@Autowired
//	private CourseJpaRepository repository;

	@Autowired
	private CourseSpringDataJpaRepository repository;

	@Override
	public void run(String... args) throws Exception {
		repository.save(new Course(1, "Learn AWS Jpa!", "in28minutes"));
		repository.save(new Course(2, "Learn Azure Jpa!", "in28minutes"));
		repository.save(new Course(3, "Learn DevOps Jpa!", "in28minutes"));

		repository.deleteById(1l);

		System.out.println(repository.findById(2l));
		System.out.println(repository.findById(3l));
	}

}
```

Exploring Spring Data JPA Further: Built-in and Custom Query Methods

1. Finding All Entities

To retrieve all courses from the database, use the findAll() method provided by JpaRepository.

```
System.out.println(repository.findAll());
```

- **Explanation:**
  - findAll() returns a list of all Course entities in the database.
  - This triggers a SQL query similar to SELECT * FROM course.

2. Counting Entities

To count the number of courses in the table, use the count() method.

```
System.out.println(repository.count());
```

- **Explanation:**
  - count() returns the number of Course records.
  - This triggers a SQL query similar to SELECT COUNT(*) FROM course.

3. Defining Custom Finder Methods (Derived Queries)

Spring Data JPA allows you to define custom query methods in your repository interface by following a naming convention.

Repository Interface Example:

CourseSpringDataJpaRepository.java

```
package com.in28minutes.springboot.learn_jpa_and_hibernate.course.springdatajpa;

import java.util.List;

import org.springframework.data.jpa.repository.JpaRepository;

import com.in28minutes.springboot.learn_jpa_and_hibernate.course.Course;

public interface CourseSpringDataJpaRepository extends JpaRepository<Course, Long>{

	List<Course> findByAuthor(String author);
	List<Course> findByName(String name);


}
```

- **Explanation:**
  - findByAuthor(String author) will find all Course records where the author field matches the input.
  - findByName(String name) will find all Course records where the name field matches the input.

- The method names follow the pattern findBy<FieldName>, where <FieldName> matches an attribute in the Course entity.

4. Using Custom Finder Methods

You can use these methods directly in your code, just like the built-in ones.

```
System.out.println(repository.findByAuthor("in28minutes")); // Returns courses authored by "in28minutes"
System.out.println(repository.findByAuthor("")); // Returns courses authored by ""

System.out.println(repository.findByName("Learn AWS Jpa!")); // Returns courses named "Learn AWS Jpa!"
System.out.println(repository.findByName("Learn DevOps Jpa!")); // Returns courses named "Learn DevOps Jpa!"
```

- **Explanation:**
  - Spring Data JPA automatically creates the correct SQL queries for these method calls.
  - If no record matches, you get an empty list.

5. Complete Example: CommandLineRunner

```
package com.in28minutes.springboot.learn_jpa_and_hibernate.course;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

import
com.in28minutes.springboot.learn_jpa_and_hibernate.course.springdatajpa.CourseSpringDataJpaR
epository;

@Component
public class CourseCommandLineRunner implements CommandLineRunner {

//	@Autowired
//	private CourseJdbcRepository repository;

//	@Autowired
//	private CourseJpaRepository repository;

	@Autowired
	private CourseSpringDataJpaRepository repository;

	@Override
	public void run(String... args) throws Exception {
		repository.save(new Course(1, "Learn AWS Jpa!", "in28minutes"));
		repository.save(new Course(2, "Learn Azure Jpa!", "in28minutes"));
		repository.save(new Course(3, "Learn DevOps Jpa!", "in28minutes"));

		repository.deleteById(1l);

		System.out.println(repository.findById(2l));
		System.out.println(repository.findById(3l));
```

```
                System.out.println(repository.findAll());
                System.out.println(repository.count());

                System.out.println(repository.findByAuthor("in28minutes"));
                System.out.println(repository.findByAuthor(""));

                System.out.println(repository.findByName("Learn AWS Jpa!"));
                System.out.println(repository.findByName("Learn DevOps Jpa!"));

        }

}
```

- **Explanation:**
  - Three courses are saved.
  - The course with ID 1 is deleted.
  - Courses with IDs 2 and 3 are fetched and printed.
  - All remaining courses are printed.
  - The total count of courses is displayed.
  - Courses are searched by author and by name using the custom finder methods.

```
Hibernate: select c1_0.id,c1_0.author,c1_0.name from course c1_0 where c1_0.id=?
Hibernate: insert into course (author,name,id) values (?,?,?)
Hibernate: select c1_0.id,c1_0.author,c1_0.name from course c1_0 where c1_0.id=?
Hibernate: insert into course (author,name,id) values (?,?,?)
Hibernate: select c1_0.id,c1_0.author,c1_0.name from course c1_0 where c1_0.id=?
Hibernate: insert into course (author,name,id) values (?,?,?)
Hibernate: select c1_0.id,c1_0.author,c1_0.name from course c1_0 where c1_0.id=?
Hibernate: delete from course where id=?
Hibernate: select c1_0.id,c1_0.author,c1_0.name from course c1_0 where c1_0.id=?
Optional[Course [id=2, name=Learn Azure Jpa!, author=in28minutes]]
Hibernate: select c1_0.id,c1_0.author,c1_0.name from course c1_0 where c1_0.id=?
Optional[Course [id=3, name=Learn DevOps Jpa!, author=in28minutes]]
Hibernate: select c1_0.id,c1_0.author,c1_0.name from course c1_0
[Course [id=2, name=Learn Azure Jpa!, author=in28minutes], Course [id=3, name=Learn DevOps Jpa!, author=in28minutes]]
Hibernate: select count(*) from course c1_0
2
Hibernate: select c1_0.id,c1_0.author,c1_0.name from course c1_0 where c1_0.author=?
[Course [id=2, name=Learn Azure Jpa!, author=in28minutes], Course [id=3, name=Learn DevOps Jpa!, author=in28minutes]]
Hibernate: select c1_0.id,c1_0.author,c1_0.name from course c1_0 where c1_0.author=?
[]
Hibernate: select c1_0.id,c1_0.author,c1_0.name from course c1_0 where c1_0.name=?
[]
Hibernate: select c1_0.id,c1_0.author,c1_0.name from course c1_0 where c1_0.name=?
[Course [id=3, name=Learn DevOps Jpa!, author=in28minutes]]
```

6. How Custom Query Methods Work

- Spring Data JPA parses method names like findByAuthor and findByName and automatically generates the appropriate SQL.
- No manual query writing or implementation is required—just declare the method in your repository interface.

7. Summary

- findAll() and count() are built-in methods provided by JpaRepository.
- Custom queries can be defined by simply following the naming conventions (findBy<FieldName>).
- Spring Data JPA automatically implements these methods and generates the SQL.
- Your repository can grow with more custom queries as needed, without additional boilerplate code.

This demonstrates the full power and convenience of Spring Data JPA: minimal code,

maximum functionality, and no SQL writing for common queries!

Hibernate vs JPA: Understanding the Difference

1. What is JPA?

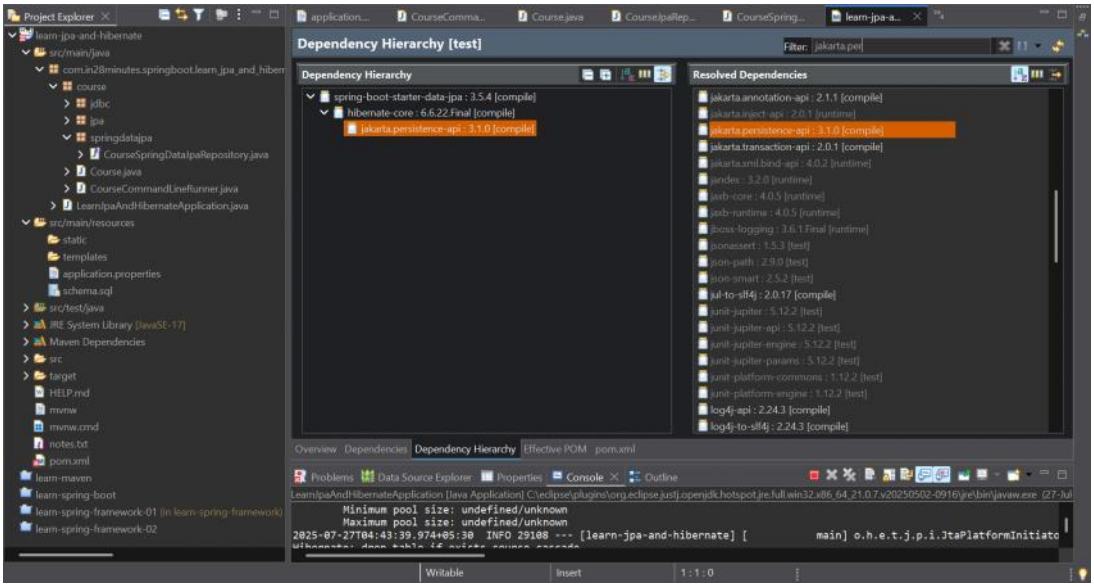
- **JPA (Java Persistence API)** is a **specification**—an API that defines how Java objects (entities) are mapped to relational database tables.
- JPA is like an **interface**: it outlines what you can do (define entities, map attributes, manage entities), but **does not provide the actual implementation.**
- **JPA Annotations** come from jakarta.persistence:
  - @Entity, @Id, @Column, etc.
  - Example usage in code:

```
import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.Column;

@Entity
public class Course {
    @Id
    private long id;
    @Column
    private String name;
    // ...
}
```

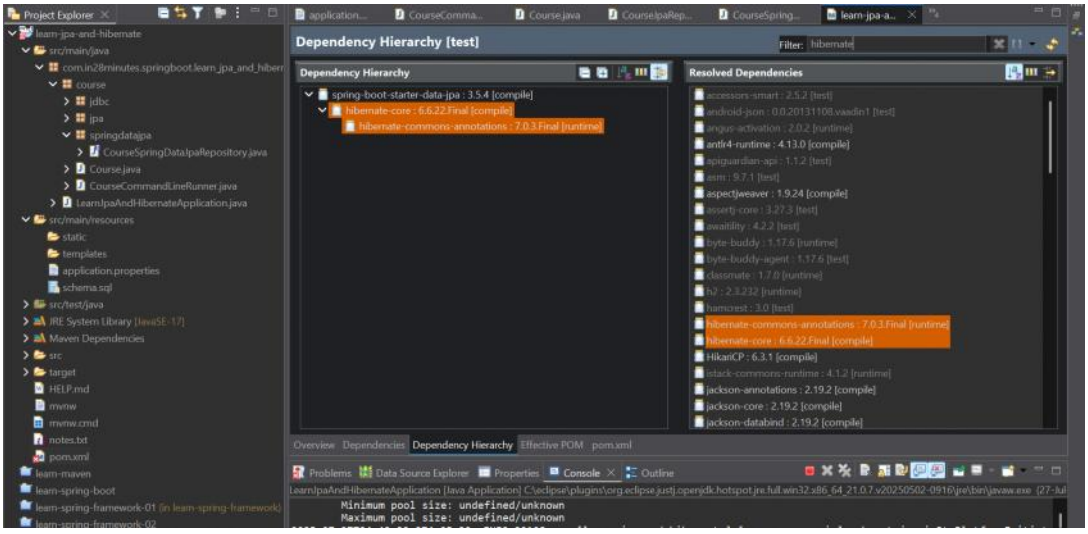
- The **EntityManager** API is also defined by JPA for managing entities.

2. What is Hibernate?

- **Hibernate** is a **popular implementation of the JPA specification.**
- Hibernate provides the actual code that runs when you use JPA APIs.
- When you use Spring Boot with spring-boot-starter-data-jpa, both **JPA** (the specification) and **Hibernate** (the implementation) are brought into your project via Maven dependencies:
  - jakarta.persistence-api (JPA)



- hibernate-core (Hibernate)



- **Hibernate provides its own annotations** (like org.hibernate.annotations.Entity), but these are specific to Hibernate.

3. How Do JPA and Hibernate Work Together?

- In your code, you typically use only **JPA annotations and APIs** (from jakarta.persistence).
- Hibernate is added as a dependency and acts **behind the scenes as the implementation**.
- This means your code is **not tightly coupled to Hibernate** and can work with any other JPA implementation (like EclipseLink or Toplink) with minimal changes.

4. Why Not Use Hibernate Directly?

- Using Hibernate-specific annotations or APIs (from org.hibernate.annotations) will make your code **dependent on Hibernate**.
- This is called **vendor lock-in**: if you want to switch to another JPA implementation in the future, it would be harder.
- By sticking to JPA annotations and APIs, you keep your code **portable** and **implementation-agnostic**.

5. Summary Table (from slides)

Aspect	JPA	Hibernate
What is it?	Specification/API	Implementation of JPA
Defines entities/attributes mapping	Yes	Follows JPA spec, adds extra
Who manages the entities?	EntityManager (API)	EntityManager (implementation)
Annotations used	jakarta.persistence.*	org.hibernate.annotations.*
Other implementations?	Yes (e.g., Toplink)	N/A
Vendor lock-in?	No	Yes, if using Hibernate APIs

6. Best Practice

- **Always use JPA annotations and APIs in your code.**
- Let Spring Boot and Hibernate handle the implementation details.
- This gives you flexibility to switch JPA providers and keeps your code clean and standard.

7. Key Takeaway

- **JPA** is the **specification** ("what you can do").
- **Hibernate** is an **implementation** ("how it's actually done").
- Use **JPA in your code**, and let Hibernate (or any other implementation) do the heavy lifting in the background.

What to Do When You Face a Challenge or Get Stuck

1. Try Harder and Troubleshoot Yourself

- Make a genuine effort to solve the problem on your own first.
- Resist the urge to immediately seek help.
- Troubleshooting independently builds resilience and deepens your understanding.

2. Ask Google

- Enter your error message or question into Google.
- Many others have likely faced similar issues, and you can often find solutions or explanations online.

3. Ask ChatGPT or Other Generative AI Tools

- Use tools like ChatGPT to get explanations, code snippets, and debugging tips.
- Make sure your question is clear and concise for the best results.
- Remember:
  - ChatGPT isn't always accurate.
  - Don't share sensitive or personal information.

4. Ask for Help in the Course Q&A or Discussion

Board

- If the previous steps don't work, post your question in the course Q&A or discussion forum.
- The instructor, team, and fellow students are there to help.
- Provide as many details as possible to make it easier for others to assist you.

5. Give Back: Help Others

- If you see questions you can answer, share your knowledge.
- Teaching and helping others deepens your own learning and strengthens the community.

6. Take a Break if Needed

- If you're still stuck after trying everything, step away for a while.
- Take a walk, rest, or sleep. Sometimes solutions come more easily after a break.

7. Encouragement

- Progress is important and you're doing great—keep going!

Section 13: Build Java Web Application with Spring Framework, Spring Boot and Hibernate

27 July 2025 05:11

Building Your First Web Application with Spring Boot

1. Web Application Concepts to Understand

- **How browsers work**
- **HTML & CSS**
- **HTTP Requests and Responses**
- **Forms and Form Submission**
- **Sessions and Authentication**

2. Spring MVC Concepts

- **Dispatcher Servlet**: The front controller that handles all requests.
- **View Resolvers**: Decide which view (page) to render.
- **Model-View-Controller (MVC) Pattern**
- **Validations**: Ensuring user input is correct.

3. Spring Boot Essentials

- **Starters**: Bundled dependencies to get started quickly (e.g., web, JPA, security).
- **Auto Configuration**: Spring Boot automatically configures components based on your dependencies.

4. Frameworks and Tools to Integrate

- **JSP (JavaServer Pages)** and **JSTL (JavaServer Pages Standard Tag Library)**
- **JPA (Java Persistence API)** for database operations
- **Bootstrap** for responsive CSS styling
- **Spring Security** for authentication and authorization
- **Databases**: MySQL and H2 (in-memory database)

5. Goal of This Section

- **Build a Todo Management Web Application** using modern Spring Boot practices.
- **Explore all relevant concepts** in a hands-on, step-by-step manner.

6. Features of the Application to Build

- User login with user ID and password

- Welcome page after login
- Todo management page (view, add, update, delete todos)
- Select and submit target dates for todos
- Fully featured navigation bar
- Logout functionality

7. Learning Approach

- Concepts will be introduced and explained **step by step**
- The focus is on **making complex ideas simple** through hands-on practice

Summary:
Building your first web application involves learning about web fundamentals, the Spring MVC pattern, Spring Boot's auto configuration, and integrating key frameworks and tools. The goal is to build a fully functional todo management app, learning everything hands-on and step by step.

Creating a Spring Boot Project Using Spring Initializr

1. Easiest Way to Create a Spring Boot Project

- Use **Spring Initializr** at start.spring.io
- It allows you to select project settings, dependencies, and download a ready-to-import project.

2. Project Setup Choices

- **Build Tool:** Maven
- **Language:** Java
- **Spring Boot Version:**
  - Use the latest **released** version of Spring Boot 3 (do **not** use snapshot versions).
  - If no release is available, use a milestone version (e.g., 3.0.0 M3).
- **Java Version:**
  - Spring Boot 3 requires Java 17 or above.
  - Recommended to use the latest Java version available (e.g., Java 18, 20, or newer).
- **Group ID:**
  - Example: com.in28minutes.springboot
- **Artifact ID:**
  - Example: myfirstwebapp

3. Selecting Dependencies

- **Spring Web:**
  - Required for building web applications and RESTful APIs using Spring MVC.
  - Uses Apache Tomcat as the default embedded server.
- **Spring Boot DevTools:**
  - For fast application restarts and live reloads during development.

4. Download and Extract the Project

- Click **Generate** to download a ZIP file of your project.
- Extract the ZIP file to a folder on your computer.

5. Importing the Project into Eclipse

- Open Eclipse and select a workspace.
- Go to **File > Import**.
- Choose **Existing Maven Projects** (search "Maven" if needed).
- Click **Next**.
- Browse and select the folder where you extracted the ZIP file.
- Select the project and click **Finish**.

6. First-Time Download Note

- The first build and dependency download may take 5-10 minutes if you haven't used this Spring Boot version before.

7. Summary

- In this step, you used Spring Initializr to create a Spring Boot project.
- You imported the Maven project into Eclipse and are now ready to start developing your web application.

Reviewing the Key Files in Your Spring Boot Project

1. Project Structure After Import

- After importing the project from Spring Initializr, you'll see these main folders:
  - `src/main/java` — Main application source code
  - `src/main/resources` — Application resources (like configuration files)
  - `src/test/java` — Test source code
  - Other files such as `pom.xml` in the project root

2. Important Files to Know

a. MyfirstwebappApplication.java

- Located in `src/main/java/com/in28minutes/springboot/myfirstwebapp/`
- This is the main class with the `main()` method.
- Used to **launch the Spring Boot application.**
- You can run this file (right-click → Run As → Java Application).

- When started, by default, the application runs on port **8080**.

b. application.properties

- Located in src/main/resources/
- Used for **application configuration** (e.g., server port, database settings, etc.).
- Example: To change the server port to 8081, add:
  server.port=8081
- Commenting out this line will revert the server to the default port 8080.
- Any settings related to your Spring Boot app can be configured here.

c. pom.xml

- Found in the project root.
- **Manages dependencies** for your project.
- Dependencies you selected in Spring Initializr (like Spring Web, DevTools) are listed here:
  - spring-boot-starter-web (for web applications)
  - spring-boot-devtools (for fast reload during development)
  - spring-boot-starter-test (for testing with JUnit, Mockito, etc.)
- Also specifies the **Java version** to use.

3. General Recommendations and Warnings

- **Spring Boot Version:**
  - Always use the **latest released version** (avoid milestone, snapshot versions like M1, M2, M3, SNAPSHOT).
- **Java Version:**
  - Use **Java 17 or newer** (Spring Boot 3.0+ requires Java 17+).
- **Eclipse IDE:**
  - Use the **latest version of Eclipse Java EE IDE** for best compatibility.

4. Summary

- **MyfirstwebappApplication.java** — Entry point to start the application.
- **application.properties** — Where you set application-level configurations.
- **pom.xml** — Where all project dependencies are managed.

5. Next Step

- Now that you're familiar with the project files, the next step is to start writing code—beginning with a HelloWorldController.

Creating Your First Hello World Controller in Spring Boot

1. Goal

- Create a URL (/say-hello) in your Spring Boot web application that returns the text:
  "Hello! What are you learning today?"

2. Create the Controller Class

- Create a new Java class named SayHelloController inside the package com.in28minutes.springboot.myfirstwebapp.hello.
- Ensure this package is a **subpackage** of your main application's package (for component scanning to work).

3. Code Explanation

```
package com.in28minutes.springboot.myfirstwebapp.hello;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class SayHelloController {

	// "say-hello" => "Hello! What are you learning today?"

	// say-hello
	// http://localhost:8080/say-hello
	@RequestMapping("say-hello")
	@ResponseBody
	public String sayHello() {
		return "Hello! What are you learning today?";
	}
}
```

a. @Controller

- Marks the class as a Spring MVC controller (a web component that handles HTTP requests).

b. @RequestMapping("say-hello")

- Maps the URL /say-hello to the sayHello() method.

c. @ResponseBody

- Tells Spring to **send the returned string directly as the HTTP response body**.
- Without this annotation, Spring would look for a view (like a JSP) named "Hello! What are you learning today?", which would cause an error.

d. Method Implementation

- public String sayHello() returns the string you want to show in the browser.

4. How to Access

- Start your Spring Boot application (run the main class).
- Open a browser and go to:
  http://localhost:8080/say-hello
- You should see:
  Hello! What are you learning today?

5. Common Issues and Tips

- Ensure your controller is in a subpackage of your main application class for Spring

Boot's component scanning to detect it.

- Make sure you have the correct imports:
  - @Controller from org.springframework.stereotype.Controller
  - @RequestMapping from org.springframework.web.bind.annotation.RequestMapping
  - @ResponseBody from org.springframework.web.bind.annotation.ResponseBody
- If you get a 404 or Whitelabel Error Page, check:
  - The URL matches the @RequestMapping value.
  - The server is running on the expected port (default is 8080).

6. Summary

- You created a simple controller that maps a URL to a method.
- You used @Controller, @RequestMapping, and @ResponseBody annotations.
- You returned a string response directly to the browser.

This is your first step in building web applications with Spring Boot!

```
package com.in28minutes.springboot.myfirstwebapp.hello;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class SayHelloController {

	// "say-hello" => "Hello! What are you learning today?"

	// say-hello
	// http://localhost:8080/say-hello
	@RequestMapping("say-hello")
	@ResponseBody
	public String sayHello() {
		return "Hello! What are you learning today?";
	}
}
```

http://localhost:8080/say-hello

Hello! What are you learning today?

Returning Raw HTML from a Controller in Spring Boot

1. Goal

- Learn how to return HTML content directly from a Spring Boot controller.
- Understand why manually constructing HTML in code becomes complex and why using views is preferred in real web applications.

2. Code Example

```
package com.in28minutes.springboot.myfirstwebapp.hello;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class SayHelloController {

	// "say-hello" => "Hello! What are you learning today?"

	// say-hello
	// http://localhost:8080/say-hello
	@RequestMapping("say-hello")
	@ResponseBody
	public String sayHello() {
		return "Hello! What are you learning today?";
	}

	@RequestMapping("say-hello-html")
	@ResponseBody
	public String sayHelloHtml() {
		StringBuffer sb = new StringBuffer();
		sb.append("<html>");
		sb.append("<head>");
		sb.append("<title> My first HTML Page - Changed</title>");
		sb.append("</head>");
		sb.append("<body>");
		sb.append("My first html page with body - Changed");
		sb.append("</body>");
		sb.append("</html>");

		return sb.toString();
	}
}
```

3. Explanation

- **@Controller:** Marks the class as a Spring MVC Controller.
- **@RequestMapping("say-hello-html"):** Maps the /say-hello-html URL to the sayHelloHtml() method.
- **@ResponseBody:** Returns the method's return value directly as the HTTP response body, instead of resolving it as a view name.
- **Building HTML with StringBuffer:**
  - The method constructs a simple HTML page as a string using StringBuffer and returns it.
  - The HTML includes <html>, <head>, <title>, and <body> tags, with a customized title and body message.

4. How It Works

- When you visit http://localhost:8080/say-hello-html, the browser displays:
  - A page titled "My first HTML Page - Changed"
  - The body says "My first html page with body - Changed"

- You can change the title or body text in the code, save, and refresh the browser to see the updates.

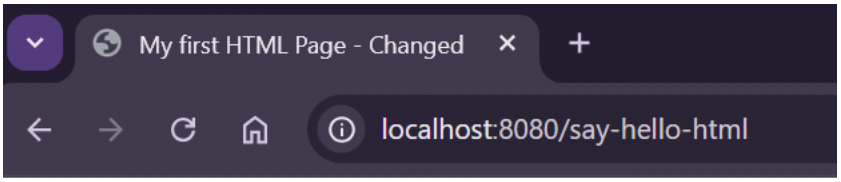
5. Key Insights

- **Manually constructing HTML in Java code quickly becomes complex and hard to maintain**, especially for larger pages.
- For real-world applications, it is better to use **views/templates** (like JSP, Thymeleaf, etc.) to manage HTML content.

6. Summary

- You learned how to return raw HTML from a controller using @ResponseBody and a string.
- This approach is simple for tiny examples, but not scalable for complex web pages.
- In the next steps, you will learn how to use views to manage HTML more efficiently.

```java
package com.in28minutes.springboot.myfirstwebapp.hello;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class SayHelloController {

	// "say-hello" => "Hello! What are you learning today?"

	// say-hello
	// http://localhost:8080/say-hello
	@RequestMapping("say-hello")
	@ResponseBody
	public String sayHello() {
		return "Hello! What are you learning today?";
	}

	@RequestMapping("say-hello-html")
	@ResponseBody
	public String sayHelloHtml() {
		StringBuffer sb = new StringBuffer();
		sb.append("<html>");
		sb.append("<head>");
		sb.append("<title> My first HTML Page - Changed</title>");
		sb.append("</head>");
		sb.append("<body>");
		sb.append("My first html page with body - Changed");
		sb.append("</body>");
		sb.append("</html>");

		return sb.toString();
	}
}
```



My first html page with body - Changed

Returning a JSP View from a Spring Boot Controller

1. Goal

- Move from returning raw HTML in the controller to using a **JSP view** for rendering HTML pages.
- Understand how Spring Boot maps controller methods to JSP files for web UI.

2. Controller Code

```
package com.in28minutes.springboot.myfirstwebapp.hello;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class SayHelloController {

	// "say-hello" => "Hello! What are you learning today?"

	// say-hello
	// http://localhost:8080/say-hello
	@RequestMapping("say-hello")
	@ResponseBody
	public String sayHello() {
		return "Hello! What are you learning today?";
	}

	@RequestMapping("say-hello-html")
	@ResponseBody
	public String sayHelloHtml() {
		StringBuffer sb = new StringBuffer();
		sb.append("<html>");
		sb.append("<head>");
		sb.append("<title> My first HTML Page - Changed</title>");
		sb.append("</head>");
		sb.append("<body>");
		sb.append("My first html page with body - Changed");
```

```
                sb.append("</body>");
                sb.append("</html>");

                return sb.toString();
        }

        //
        // "say-hello-jsp" => sayHello.jsp
        // /src/main/resources/META-INF/resources/WEB-INF/jsp/sayHello.jsp
        // /src/main/resources/META-INF/resources/WEB-INF/jsp/welcome.jsp
        // /src/main/resources/META-INF/resources/WEB-INF/jsp/login.jsp
        // /src/main/resources/META-INF/resources/WEB-INF/jsp/todos.jsp
        @RequestMapping("say-hello-jsp")
        public String sayHelloJsp() {
                return "sayHello";
        }
}
```

Key Points:

- The method sayHelloJsp() **does not use** @ResponseBody.
- Returning the string "sayHello" tells Spring MVC to look for a JSP file named sayHello.jsp (resolved via prefix/suffix).

3. JSP File Location and Content

- **JSP path:**
  /src/main/resources/META-INF/resources/WEB-INF/jsp/sayHello.jsp
- **Content:**

```
<html>
        <head>
                <title>My First HTML Page - JSP</title>
        </head>
        <body>
                My first html page with body - JSP
        </body>
</html>
```

- **Why this location?**
  Spring Boot with embedded Tomcat expects JSP files in /META-INF/resources/WEB-INF/jsp/ under src/main/resources.

4. Spring Boot Configuration (application.properties)

```
spring.application.name=myfirstwebapp
# server.port=8081
# sayHello.jsp
spring.mvc.view.prefix=/WEB-INF/jsp/
spring.mvc.view.suffix=.jsp
logging.level.org.springframework=debug
```

- **Prefix/Suffix:**
  - When returning "sayHello" from the controller, Spring resolves it to /WEB-INF/jsp/sayHello.jsp.
- **Debug Logging:**
  - Enables detailed logs to help debug view resolution and controller mapping.

5. Maven Dependency for JSP Support (pom.xml)

```
<dependency>
    <groupId>org.apache.tomcat.embed</groupId>
    <artifactId>tomcat-embed-jasper</artifactId>
    <scope>provided</scope>
</dependency>
```

- **Why needed?**
  - tomcat-embed-jasper is required for JSP compilation in embedded Tomcat.

6. How the Mapping Works

- **Request Flow:**
  1. User visits /say-hello-jsp.
  2. Spring Boot routes to SayHelloController.sayHelloJsp().
  3. Method returns "sayHello".
  4. View Resolver adds prefix/suffix: /WEB-INF/jsp/sayHello.jsp.
  5. JSP is rendered and sent to the browser.

7. Your Own Notes (Included)

- JSP file is at:
  /src/main/resources/META-INF/resources/WEB-INF/jsp/sayHello.jsp
- Mapping:
  /say-hello-jsp → SayHelloController.sayHelloJsp → "sayHello"
  → /WEB-INF/jsp/sayHello.jsp

8. Summary Table

URL	Controller Method	Returned View Name	Resolved JSP Path
/say-hello	sayHello()	(ResponseBody)	Text Response
/say-hello-html	sayHelloHtml()	(ResponseBody)	Raw HTML String
/say-hello-jsp	sayHelloJsp()	sayHello	/WEB-INF/jsp/sayHello.jsp

9. Why Use JSP/View Templates?

- Writing HTML in Java is messy and unscalable.
- JSP files allow you to separate your HTML from Java code, making pages easier to edit and maintain.

You have now learned to return a JSP view from a Spring Boot controller, configure view resolution, and add the necessary dependencies and files. This is the recommended approach for maintainable web UI with Spring Boot + JSP.

SayHelloController.java

```
package com.in28minutes.springboot.myfirstwebapp.hello;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class SayHelloController {
```

```
	// "say-hello" => "Hello! What are you learning today?"

	// say-hello
	// http://localhost:8080/say-hello
	@RequestMapping("say-hello")
	@ResponseBody
	public String sayHello() {
		return "Hello! What are you learning today?";
	}

	@RequestMapping("say-hello-html")
	@ResponseBody
	public String sayHelloHtml() {
		StringBuffer sb = new StringBuffer();
		sb.append("<html>");
		sb.append("<head>");
		sb.append("<title> My first HTML Page - Changed</title>");
		sb.append("</head>");
		sb.append("<body>");
		sb.append("My first html page with body - Changed");
		sb.append("</body>");
		sb.append("</html>");

		return sb.toString();
	}

	//
	// "say-hello-jsp" => sayHello.jsp
	// /src/main/resources/META-INF/resources/WEB-INF/jsp/sayHello.jsp
	// /src/main/resources/META-INF/resources/WEB-INF/jsp/welcome.jsp
	// /src/main/resources/META-INF/resources/WEB-INF/jsp/login.jsp
	// /src/main/resources/META-INF/resources/WEB-INF/jsp/todos.jsp
	@RequestMapping("say-hello-jsp")
	public String sayHelloJsp() {
		return "sayHello";
	}
}
```

sayHello.jsp

```
<html>
	<head>
		<title>My First HTML Page - JSP</title>
	</head>
	<body>
		My first html page with body - JSP
	</body>
</html>
```

```
application.properties
spring.application.name=myfirstwebapp
# server.port=8081
# sayHello.jsp
spring.mvc.view.prefix=/WEB-INF/jsp/
spring.mvc.view.suffix=.jsp
logging.level.org.springframework=debug
```

pom.xml

```
<dependency>
     <groupId>org.apache.tomcat.embed</groupId>
     <artifactId>tomcat-embed-jasper</artifactId>
```

```
    <scope>provided</scope>
</dependency>
#Check java version if facing any error change it from java 17 to java 21
```

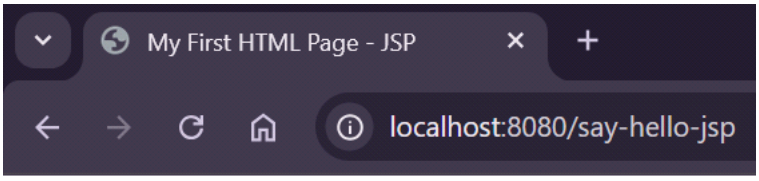
The application initially crashed with a JVM EXCEPTION_ACCESS_VIOLATION when accessing JSP pages, but after changing the Java version from 17 to 21 in pom.xml to match the runtime environment, it now works successfully and returns HTTP 200 OK.

JSP

/src/main/resources/META-INF/resources/WEB-INF/jsp/sayHello.jsp

/say-hello-jsp => SayHelloController - sayHelloJsp method => sayHello

/WEB-INF/jsp/sayHello.jsp



Adding a Login Page with Spring Boot, Controllers, and JSP

1. Goal

- Map the URL /login to a controller.
- Display a custom login.jsp page when /login is accessed.
- Understand how easy it is to add new views and controllers after initial configuration.

2. Steps Taken

a. Create a Login Controller

- **Package:**
  Place the controller in the package:
  com.in28minutes.springboot.myfirstwebapp.login
- **Controller Code:**

```
package com.in28minutes.springboot.myfirstwebapp.hello.login;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class LoginController {
	// /login => com.in28minutes.springboot.myfirstwebapp.login.LoginController =>
login.jsp

	@RequestMapping("login")
	public String gotoLoginPage() {
		return "login";
	}
}
```

- **@Controller**: Tells Spring this class is a web controller.

- **@RequestMapping("login"):** Maps /login URL to the gotoLoginPage() method.
- **Return value "login":** Spring resolves this to login.jsp (using the configured prefix and suffix).

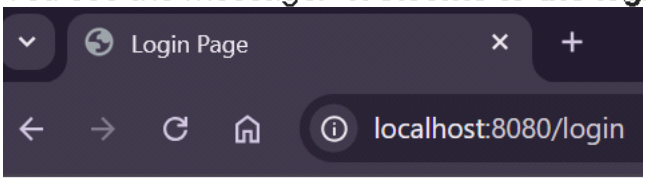
b. Create the JSP View

- **Location:**
  /src/main/resources/META-INF/resources/WEB-INF/jsp/login.jsp
- **Content:**

```
<html>
	<head>
		<title>Login Page</title>
	</head>
	<body>
		Welcome to the login Page!
	</body>
</html>
```

c. Test the Flow

- Go to http://localhost:8080/login in your browser.
- The LoginController handles the request and returns "login".
- Spring resolves this to /WEB-INF/jsp/login.jsp.
- You see the message: **"Welcome to the login Page!"**



Welcome to the login Page!

3. Troubleshooting Tips

- If you see "Resource not found" for /login, make sure your controller:
  - Has the correct @Controller annotation.
  - Is in a package scanned by Spring Boot (usually a subpackage of your main class).
- If you see "Login JSP not found," make sure the JSP file exists in the correct location.

4. Key Takeaways

- Once your project is set up with JSP support and proper configuration, **adding new pages is easy:**
  - Add a new controller method for the new URL.
  - Create the corresponding JSP file.
- All previous changes to pom.xml and application.properties for JSP support are **one-time setup.**
- You can now focus on creating controllers and views (JSPs) for each page in your app.

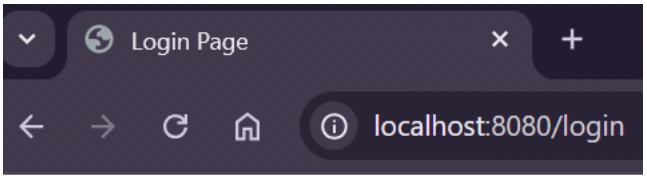
5. Summary Table

URL Path	Controller Method	Returned View Name	Resolved JSP Path
/login	LoginController.gotoLoginPage()	"login"	/WEB-INF/jsp/login.jsp

You have now added a login page using Spring Boot and JSP, and learned the general flow

for adding new views to your web application!

```
LoginController.java
package com.in28minutes.springboot.myfirstwebapp.hello.login;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class LoginController {
	// /login => com.in28minutes.springboot.myfirstwebapp.login.LoginController =>
login.jsp

	@RequestMapping("login")
	public String gotoLoginPage() {
		return "login";
	}
}

login.jsp
<html>
	<head>
		<title>Login Page</title>
	</head>
	<body>
		Welcome to the login Page!
	</body>
</html>
```



Welcome to the login Page!

Understanding Requests, Responses, and Browser DevTools (Say Hello Endpoints)

What We Did

- Tested three endpoints:
  - /say-hello: returns plain text from controller (using @ResponseBody)
  - /say-hello-html: returns raw HTML string from controller
  - /say-hello-jsp: returns a JSP view rendered as HTML

Observations in the Browser

- **View Page Source**
  - /say-hello shows plain text (no HTML structure).
  - /say-hello-html and /say-hello-jsp show proper HTML markup in the source.
- **Editing JSP**
  - Updated sayHello.jsp to include headings:
    - <h1>Heading 1</h1>
    - <h2>Heading 2</h2>
  - Refresh shows the headings rendered by the browser, confirming HTML is returned and interpreted.

Using Chrome DevTools (Inspect → Network)

- Filter by "Doc" to view page/document requests.
- **Example: Success request to /say-hello-jsp**
  - Request URL: http://localhost:8080/say-hello-jsp

    - Request Method: GET
    - Status Code: 200 OK
    - Tabs:
        - Headers: shows Request headers (Host, User-Agent, etc.) and Response headers (Content-Type: text/html; charset=UTF-8, etc.)
        - Response: shows the exact HTML returned.
- **Example: Error request to non-existing URL (/say-hello-jsp-1)**
    - Status Code: 404 Not Found
    - The browser displays an error page; Network tab confirms the 404 status.

Key HTTP Concepts Reinforced

- **HTTP Request**
    - Triggered by browser actions (typing URL, clicking links, form submissions).
    - Methods: We used GET here; later you will see POST and others.
- **HTTP Response**
    - Server returns a response to the browser.
    - Commonly Content-Type: text/html for web pages (could be JSON, etc., for APIs).
    - Includes a Status Code indicating success or error:
        - 200: Success
        - 404: Resource not found
- **End-to-End Flow**
    - Browser sends HTTP request → Spring Boot web app handles it → returns HTTP response → browser renders the result.

Endpoints Recap

- **/say-hello**
    - Simple text response from controller.
    - Browser source shows plain text.
- **/say-hello-html**
    - Raw HTML string returned via controller.
    - Browser source shows HTML markup.
- **/say-hello-jsp**
    - Controller returns view name; JSP view rendered to HTML.
    - Editing JSP (e.g., adding <h1>, <h2>) changes rendered page.

Practical Tips

- Use View Page Source to confirm what the server actually returned.
- Use DevTools Network tab to:
    - Verify request method (GET/POST).
    - Check response status (200 vs 404).
    - Inspect response headers and body.
- If an endpoint returns 404, verify:
    - Correct URL path.
    - Controller mapping exists and matches the path.
    - Application is running on the expected port.

Quick Checklist

- Can you access all three endpoints and see expected outputs?
- Do DevTools show 200 for valid URLs and 404 for invalid ones?
- Does View Source for /say-hello-html and /say-hello-jsp show HTML?
- Did adding <h1>/<h2> to the JSP reflect on page refresh?

Request Parameters and Passing Data to JSP (Using Model and ModelMap)

What We Did

- Extended the /login URL to accept a query parameter name (e.g., localhost:8080/login?

name=Ranga) to display “Welcome to login page, <name>”.

- Used @RequestParam to bind the query parameter to a controller method parameter.
- Used ModelMap to pass the parameter value from the controller to the JSP.
- Used JSP Expression Language to display the model attribute.

Key Steps and Concepts

- Request param via query string:
  - Format: ?name=Ranga appended to localhost:8080/login.
- Controller handling:
  - Add @RequestParam to bind the web request parameter to a method parameter.
  - Import: org.springframework.web.bind.annotation.RequestParam.
  - Print the value using System.out.println to verify (not recommended for production).
- Passing data to JSP:
  - Introduce Model/ModelMap to transfer data from controller to view.
  - Import: org.springframework.ui.ModelMap.
  - Use model.put("name", name) to add the attribute.
- JSP usage:
  - Use JSP Expression Language: ${name} (dollar + open brace + variable + close brace) to read the model attribute.
  - Update the page text to include the name.

Final Code (From Lecture)

- Controller:

```
package com.in28minutes.springboot.myfirstwebapp.hello.login;

import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
public class LoginController {
	// /login => com.in28minutes.springboot.myfirstwebapp.login.LoginController =>
login.jsp


	//  http://localhost:8080/login?name=Nilesh
	// Model
	@RequestMapping("login")
	public String gotoLoginPage(@RequestParam String name, ModelMap model) {
		model.put("name", name);
		System.out.println("Request Pram is " + name); // NOT RECOMMENDED FOR PRODUCTION
		return "login";
	}
}
```

- JSP (login.jsp):

```
<html>
	<head>
		<title>Login Page</title>
	</head>
	<body>
		Welcome to the login Page ${name}!
	</body>
</html>
```

Observations/Demonstration

- Entering different values in the URL (e.g., Ranga, Sathish, John, Ravi) updates the rendered message accordingly.
- Console/log prints: Request param is <Name> when the request is made.

Summary (As Stated)

- Pass query parameters to controller methods using @RequestParam.
- To show them in JSP, put values into a Model (e.g., ModelMap).
- The view (JSP) can pick them using expression language ${...}.
- Practice by modifying controller and JSP to understand behavior.

Configuring and Using Logging in Spring Boot (LoginController + application.properties)

What We Did

- Added SLF4J logging to LoginController.
- Logged messages at different levels: DEBUG, INFO, WARN.
- Continued to print with System.out.println (not recommended for production).
- Configured view resolution and logging levels in application.properties.

Controller Code

```
package com.in28minutes.springboot.myfirstwebapp.hello.login;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
public class LoginController {

	private Logger logger = LoggerFactory.getLogger(getClass());
	// /login => com.in28minutes.springboot.myfirstwebapp.login.LoginController =>
login.jsp


	// http://localhost:8080/login?name=Nilesh
	// Model
	@RequestMapping("login")
	public String gotoLoginPage(@RequestParam String name, ModelMap model) {
		model.put("name", name);
		logger.debug("Request param is {}", name);
		logger.info("I want this printed at info level");
		logger.warn("I want this printed at warn level");

		System.out.println("Request Pram is " + name); // NOT RECOMMENDED FOR PRODUCTION

		return "login";
	}
}
```

Key Points in Controller

- Logger initialization: private Logger logger = LoggerFactory.getLogger(getClass());
- Logging examples:
  - logger.debug("Request param is {}", name);
  - logger.info("I want this printed at info level");

    - logger.warn("I want this printed at warn level");
- Model usage: model.put("name", name);
- Endpoint: GET /login?name=Nilesh returns view login (resolved to JSP).

application.properties

```
spring.application.name=myfirstwebapp
# server.port=8081
# sayHello.jsp
spring.mvc.view.prefix=/WEB-INF/jsp/
spring.mvc.view.suffix=.jsp
logging.level.org.springframework=info
logging.level.com.in28minutes.springboot.myfirstwebapp=info
```

Key Points in Properties

- View resolver:
    - Prefix: /WEB-INF/jsp/
    - Suffix: .jsp
    - View name login -> /WEB-INF/jsp/login.jsp
- Logging levels:
    - org.springframework set to info
    - com.in28minutes.springboot.myfirstwebapp set to info (affects LoginController package)
- Optional server port commented out (8081).

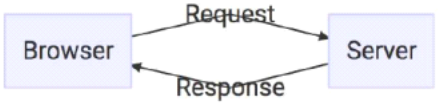
What You'll See in Logs (Given Settings)

- With logging level info for the app package:
    - INFO and WARN messages will appear.
    - DEBUG message will not appear.
- System.out.println output will always appear (separate from logging levels).

Quick Checklist

- Access /login?name=Nilesh and confirm:
    - Page renders via login.jsp.
    - INFO and WARN logs are printed.
    - DEBUG log is suppressed at current level.
    - Console shows System.out.println message.

Spring MVC Request Flow (DispatcherServlet, Controller, View Resolver, JSP)

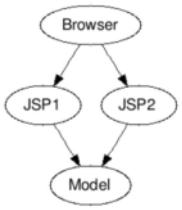


Context and History

- Early Java web apps: all code in JSPs (view logic, flow logic, database access) → hard to maintain, no separation of concerns. Called Model 1 Architecture.

Peek into History - Model 1 Arch.

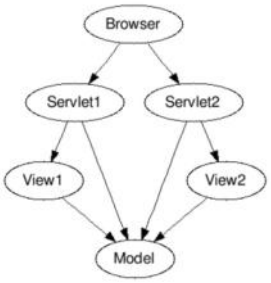
- **ALL CODE** in Views (JSPs, ...)
  - View logic
  - Flow logic
  - Queries to databases
- **Disadvantages:**
  - VERY complex JSPs
  - ZERO separation of concerns
  - Difficult to maintain



- Model 2 Architecture: clear separation:
  - Model: data used to generate the view
  - View: renders information to the user
  - Controller/Servlets: control the flow

Peek into History - Model 2 Arch.

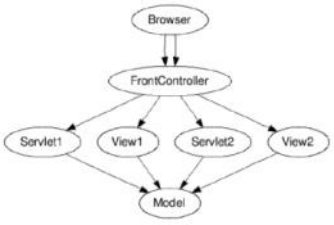
- How about separating concerns?
  - **Model**: Data to generate the view
  - **View**: Show information to user
  - **Controller**: Controls the flow
- **Advantage**: Simpler to maintain
- **Concern:**
  - Where to implement common features to all controllers?



- Model 2 with Front Controller: all requests go to a single front controller; common features (authentication/authorization) implemented centrally; forwards to appropriate controller/view.

Model 2 Architecture - Front Controller

- **Concept**: All requests flow into a central controller
  - Called as **Front Controller**
- Front Controller controls flow to Controller's and View's
  - Common features can be implemented in the Front Controller



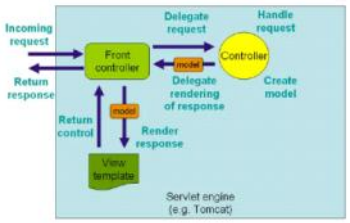
DispatcherServlet (Spring MVC Front Controller)

- Spring MVC's implementation of the Model 2 Front Controller pattern.
- Receives all requests (e.g., /login, /say-hello, /say-hello-jsp).
- Responsibilities:
  1. Identify the correct controller method based on URL.
  2. Execute the controller method.
  3. Obtain model data and view name from the controller.
  4. Resolve the view name to an actual view using ViewResolver.

5. Render the view (make model available to the view) and return HTTP response.

Spring MVC Front Controller - Dispatcher Servlet

- **A:** Receives HTTP Request
- **B:** Processes HTTP Request
  - **B1:** Identifies correct Controller method
    - Based on request URL
  - **B2:** Executes Controller method
    - Returns Model and View Name
  - **B3:** Identifies correct View
    - Using ViewResolver
  - **B4:** Executes view
- **C:** Returns HTTP Response



View Resolution

- View name (e.g., login) mapped to JSP path using prefix and suffix.
- Example mapping:
  - Prefix: /WEB-INF/jsp/
  - Suffix: .jsp
  - View name login → /WEB-INF/jsp/login.jsp
- ViewResolver performs: prefix + viewName + suffix.

End-to-End Example: localhost:8080/login

- B1: Identifies correct controller method
  - /login => LoginController.gotoLoginPage
- B2: Executes controller method
  - Puts data into Model
  - Returns view name: login
- B3: Identifies correct View
  - /WEB-INF/jsp/login.jsp
- B4: Executes View
  - Renders JSP with the provided Model and returns response to browser

High-Level Flow Recap

- Request → DispatcherServlet → Controller method → Model + View name → ViewResolver → JSP execution → HTTP response back to browser.

Building a Basic Login Form and Using POST vs GET

What We Did

- Simplified the controller to return the login view without request parameters or model.
- Created a basic HTML form on the login page to capture name and password.
- Switched the form to use HTTP POST to avoid sending credentials in the URL.

Controller (As Provided)

```
package com.in28minutes.springboot.myfirstwebapp.hello.login;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class LoginController {
	// login
	@RequestMapping("login")
	public String gotoLoginPage() {
```

```
            return "login";
        }
}
```

- Maps /login to return the view name login (resolved to login.jsp).

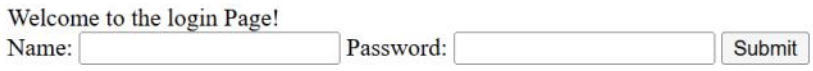
JSP View (As Provided)

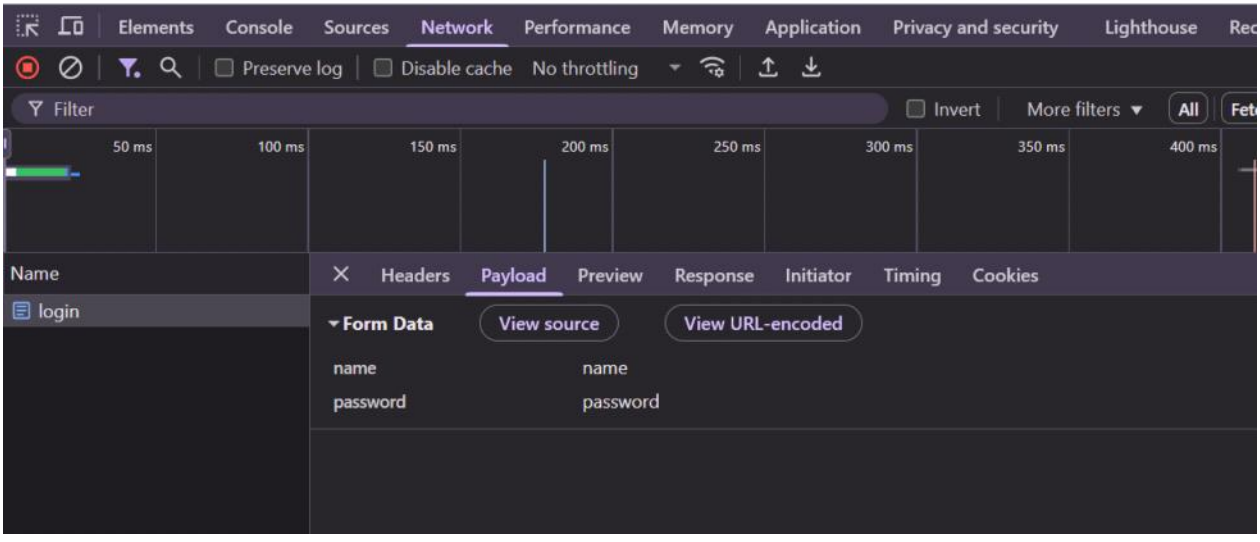
```
<html>
    <head>
        <title>Login Page</title>
    </head>
    <body>
        Welcome to the login Page!
        <form method="post">
            Name: <input type="text" name="name">
            Password: <input type="password" name="password">
            <input type="submit">
        </form>
    </body>
</html>
```

- Displays a welcome message.
- Form uses method="post".
- Captures two fields:
  - name via <input type="text" name="name">
  - password via <input type="password" name="password">
- Submit button sends the form data.

Key Observations

- With GET, form values appear in the URL query string (not secure).
- With POST, values are sent as form data (visible under "Payload" in browser DevTools Network tab), not in the URL.





Handling GET vs POST for Login and Passing Form Data to Welcome Page

What We Did

- Split /login handling into two methods:
  - GET /login → returns login view.
  - POST /login → captures form data and returns welcome view.
- Captured form fields name and password with @RequestParam.
- Passed captured values to the view using ModelMap.
- Displayed the values on the JSP using ${name} and ${password}.

Controller (As Provided)

```
package com.in28minutes.springboot.myfirstwebapp.hello.login;

import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;


@Controller
public class LoginController {
	@RequestMapping(value="login",method = RequestMethod.GET)
	public String gotoLoginPage() {
		return "login";
	}

	@RequestMapping(value="login",method = RequestMethod.POST)
	// login?name=Nilesh RequestParam
	public String gotoWelcomePage(@RequestParam String name,
			@RequestParam String password, ModelMap model) {
		model.put("name", name);
		model.put("password", password);
		return "welcome";
	}
}
```

View (As Provided)

```
<html>
	<head>
		<title>Login Page</title>
	</head>
	<body>
		<div>Welcome to in28minutes!</div>
		<div>Your Name: ${name}</div>
		<div>Your Password: ${password}</div>
	</body>
</html>
```

Flow Summary

- GET /login → renders login.jsp.
- POST /login with form data (name, password) → adds to ModelMap → returns welcome → JSP displays ${name} and ${password}.

Adding Authentication with Service, Constructor Injection, and Error Handling

What We Did

- Introduced an AuthenticationService to validate credentials.
- Used constructor injection to wire AuthenticationService into LoginController.
- Split /login into:
  - GET: render login page.
  - POST: authenticate and redirect to welcome on success; return to login with error on failure.
- Updated view to display user name on welcome page.

Controller (As Provided)

```
package com.in28minutes.springboot.myfirstwebapp.hello.login;

import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;


@Controller
public class LoginController {

	private AuthenticationService authenticationService;

	public LoginController(AuthenticationService authenticationService) {
		super();
		this.authenticationService = authenticationService;
	}

	@RequestMapping(value="login",method = RequestMethod.GET)
	public String gotoLoginPage() {
		return "login";
	}

	@RequestMapping(value="login",method = RequestMethod.POST)
	// login?name=Nilesh RequestParam
	public String gotoWelcomePage(@RequestParam String name,
			@RequestParam String password, ModelMap model) {
		model.put("name", name);

		if(authenticationService.authenticate(name, password)) {
			// Authentication
			// name - in28minutes
			// password - dummy

			return "welcome";
		}

		model.put("errorMessage", "Invalid Credentials! Please try again.");
		return "login";

	}
}
```

Welcome View (As Provided)

```
<html>
	<head>
		<title>Login Page</title>
```

```
	</head>
	<body>
		<div>Welcome to in28minutes!</div>
		<div>Your Name: ${name}</div>
	</body>
</html>
```

Authentication Service (As Provided)

```
package com.in28minutes.springboot.myfirstwebapp.hello.login;

import org.springframework.stereotype.Service;

@Service
public class AuthenticationService {

	public boolean authenticate(String username, String password) {

		boolean isValidUserName = username.equalsIgnoreCase("in28minutes");
		boolean isValidPassword = password.equalsIgnoreCase("dummy");

		return isValidUserName && isValidPassword;
	}
}
```

Flow

- GET /login → returns login.
- POST /login with name and password:
  - model.put("name", name).
  - If authenticationService.authenticate(name, password) is true → return welcome.
  - Else → add errorMessage to model and return login.

Creating Todo Model and Service with Static List

Files and Packages

- Package: com.in28minutes.springboot.myfirstwebapp.todo
- Classes: Todo, TodoService

Todo Class (Model)

- Imports: java.time.LocalDate
- Fields:
  - int id
  - String username
  - String descripton
  - LocalDate targetDate
  - boolean done
- Constructor:
  - Todo(int id, String username, String descripton, LocalDate targetDate, boolean done)
- Getters/Setters:
  - getId(), setId(int id)
  - getUsername(), setUsername(String username)
  - getDescripton(), setDescripton(String descripton)
  - getTargetDate(), setTargetDate(LocalDate targetDate)
  - isDone(), setDone(boolean done)
- toString():
  - Returns string with all fields:

- Todo [id=..., username=..., descripton=..., targetDate=..., done=...]

TodoService Class

- Imports:
  - java.time.LocalDate
  - java.util.List
- Fields:
  - private static List<Todo> todos;
- Static initializer:
  - Adds three Todo entries:
    - (1, "in28minutes", "Learn AWS", LocalDate.now().plusYears(1), false)
    - (1, "in28minutes", "Learn DevOps", LocalDate.now().plusYears(2), false)
    - (1, "in28minutes", "Learn Full Stack Development", LocalDate.now().plusYears(3), false)
- Method:
  - public List<Todo> findByUsername(String username) { return todos; }

Comments in Code

- In Todo:
  - // Database (MySQL)
  - // Static List of todos => Database(H2,MySQL)

Flow/Usage Summary

- Todo represents a to-do item with id, username, description, target date, and completion status.
- TodoService maintains a static list of Todo items and returns the list via findByUsername(String username).

```
package com.in28minutes.springboot.myfirstwebapp.todo;

import java.time.LocalDate;

// Database (MySQL)
// Static List of todos => Database(H2,MySQL)



public class Todo {

	private int id;
	private String username;
	private String descripton;
	private LocalDate targetDate;
	private boolean done;

	public Todo(int id, String username, String descripton, LocalDate targetDate, boolean
done) {
		super();
		this.id = id;
		this.username = username;
		this.descripton = descripton;
		this.targetDate = targetDate;
		this.done = done;
	}

	public int getId() {
		return id;
	}

	public void setId(int id) {
		this.id = id;
```

```
	}

	public String getUsername() {
		return username;
	}

	public void setUsername(String username) {
		this.username = username;
	}

	public String getDescripton() {
		return descripton;
	}

	public void setDescripton(String descripton) {
		this.descripton = descripton;
	}

	public LocalDate getTargetDate() {
		return targetDate;
	}

	public void setTargetDate(LocalDate targetDate) {
		this.targetDate = targetDate;
	}

	public boolean isDone() {
		return done;
	}

	public void setDone(boolean done) {
		this.done = done;
	}

	@Override
	public String toString() {
		return "Todo [id=" + id + ", username=" + username + ", descripton=" + descripton +
", targetDate=" + targetDate
				+ ", done=" + done + "]";
	}

}
```

```
package com.in28minutes.springboot.myfirstwebapp.todo;

import java.time.LocalDate;
import java.util.List;

public class TodoService {
	private static List<Todo> todos;
	static {
		todos.add(new Todo(1, "in28minutes", "Learn AWS",
				LocalDate.now().plusYears(1), false));
		todos.add(new Todo(1, "in28minutes", "Learn DevOps",
				LocalDate.now().plusYears(2), false));
		todos.add(new Todo(1, "in28minutes", "Learn Full Stack Development",
				LocalDate.now().plusYears(3), false));
	}
```

```
	public List<Todo> findByUsername(String username) {
		return todos;
	}
}
```

Todo Model, Controller, Views, and Request vs Model vs Session

Todo Domain

- Fields:
  - id
  - username
  - description
  - targetDate
  - done
- Components:
  - TodoController
  - listTodos.jsp

Request vs Model vs Session

- LoginController (with session attribute)

```
package com.in28minutes.springboot.myfirstwebapp.hello.login;

import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.SessionAttributes;

@Controller
@SessionAttributes("name")
public class LoginController {

	private AuthenticationService authenticationService;

	public LoginController(AuthenticationService authenticationService) {
		super();
		this.authenticationService = authenticationService;
	}

	@RequestMapping(value="login",method = RequestMethod.GET)
	public String gotoLoginPage() {
		return "login";
	}

	@RequestMapping(value="login",method = RequestMethod.POST)
	// login?name=Nilesh RequestParam
	public String gotoWelcomePage(@RequestParam String name,
			@RequestParam String password, ModelMap model) {
		model.put("name", name);

		if(authenticationService.authenticate(name, password)) {
			// Authentication
			// name - in28minutes
			// password - dummy
```

```
                return "welcome";
            }

            model.put("errorMessage", "Invalid Credentials! Please try again.");
            return "login";

        }
}
```

- Welcome page

```
<html>
    <head>
        <title>Login Page</title>
    </head>
    <body>
        <div>Welcome to in28minutes!</div>
        <div>Your Name: ${name}</div>
        <div><a href="list-todos">Manage</a> your todos</div>
    </body>
</html>
```

- TodoService

```
package com.in28minutes.springboot.myfirstwebapp.todo;

import java.time.LocalDate;
import java.util.ArrayList;
import java.util.List;

import org.springframework.stereotype.Service;

@Service
public class TodoService {
    private static List<Todo> todos = new ArrayList<>();
    static {
        todos.add(new Todo(1, "in28minutes", "Learn AWS",
                LocalDate.now().plusYears(1), false));
        todos.add(new Todo(1, "in28minutes", "Learn DevOps",
                LocalDate.now().plusYears(2), false));
        todos.add(new Todo(1, "in28minutes", "Learn Full Stack Development",
                LocalDate.now().plusYears(3), false));
    }

    public List<Todo> findByUsername(String username) {
        return todos;
    }
}
```

- TodoController (using session attribute name)

```
package com.in28minutes.springboot.myfirstwebapp.todo;
```

```
import java.util.List;

import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.SessionAttributes;

@Controller
@SessionAttributes("name")
public class TodoController {

	public TodoController(TodoService todoService) {
		super();
		this.todoService = todoService;
	}

	private TodoService todoService;

	@RequestMapping("list-todos")
	public String listAllTodos(ModelMap model) {
		List<Todo> todos = todoService.findByUsername("in28minutes");
		model.addAttribute("todos", todos);

		return "listTodos";
	}
}
```

- listTodos.jsp

```
<html>
	<head>
		<title>List Todos Page</title>
	</head>
	<body>
		<div>Welcome ${name}</div>
		<div>Your Todos are ${todos}</div>
	</body>
</html>
```

Adding JSTL, Iterating Todos in a Table, and Basic JSP Formatting

What we did (from the transcript)

- Improved the look and feel by displaying todos in a proper HTML table and adding simple formatting (headings, hr).

- Continued to use JSP Expression Language (EL) for simple dynamic values from the model.
- Introduced JSTL for more complex dynamic behavior, like looping over collections.
- Added JSTL dependencies to pom.xml:
  - JSTL API
  - JSTL implementation
- Stopped and restarted the server after updating pom.xml to download dependencies and clear errors.
- Imported the JSTL core tag library in JSP using taglib with the core URI.
- Used the JSTL core tag c:forEach to iterate over the dynamic list of todos and render each row in a table.
- Ensured correct HTML table structure:
  - thead contains a header row tr with th cells
  - tbody contains data rows tr with td cells
- Verified the page, corrected duplicate IDs in the service (each todo should have a different id), refreshed to see proper output.
- Applied simple formatting to JSPs:
  - Show “Welcome ${name}”
  - Use hr
  - Use headings (e.g., h1 Your Todos)
- Logged in with valid credentials (in28minutes/dummy) so the session name is available on pages.
- Acknowledged further UI work will come later with CSS frameworks (menu, login, logout styling).

```
<dependency>
     <groupId>jakarta.servlet.jsp.jstl</groupId>
     <artifactId>jakarta.servlet.jsp.jstl-api</artifactId>
</dependency>
<dependency>
       <groupId>org.glassfish.web</groupId>
       <artifactId>jakarta.servlet.jsp.jstl</artifactId>
</dependency>
```

```
<%@ taglib prefix="c" uri="jakarta.tags.core" %>
<html>
       <head>
              <title>List Todos Page</title>
       </head>
       <body>
              <div>Welcome ${name}</div>
              <hr>
              <h1>Your Todos</h1>
              <table>
                     <thead>
                           <tr>
                                  <th>id</th>
                                  <th>Description</th>
                                  <th>Target Date</th>
                                  <th>Is Done?</th>
                           </tr>
                     </thead>
                     <tbody>
                           <c:forEach items="${todos}" var="todo">
                                  <tr>
                                         <td>${todo.id}</td>
```

```
                                    <td>${todo.description}</td>
                                    <td>${todo.targetDate}</td>
                                    <td>${todo.done}</td>
                                </tr>
                            </c:forEach>
                        </tbody>
                    </table>
            </body>
</html>

<html>
            <head>
                    <title>Welcome Page</title>
            </head>
            <body>
                    <div>Welcome ${name}</div>
                    <hr>
                    <div><a href="list-todos">Manage</a> your todos</div>
            </body>
</html>
```