Palash Bajpai

**Spring Boot**

Table of Contents

[**1. What is system design?** 2](#_Toc165803653)

[1. System Design requirements 3](#_Toc165803654)

[**2. Monolithic vs Micro service architecture** 4](#_Toc165803655)

[1. Monolithic architecture 4](#_Toc165803656)

[2. Microservice architecture 5](#_Toc165803657)

[**3. System Design Trade-offs** 6](#_Toc165803658)

[1. Latency 6](#_Toc165803659)

# **What is Spring framework?**

Spring is a **powerful and comprehensive framework for Java** that simplifies the development of enterprise-grade applications. It provides infrastructure support for building robust, maintainable, and scalable applications. Spring offers a modular design, enabling developers to use only the components they need, such as dependency injection, aspect-oriented programming, or data access.

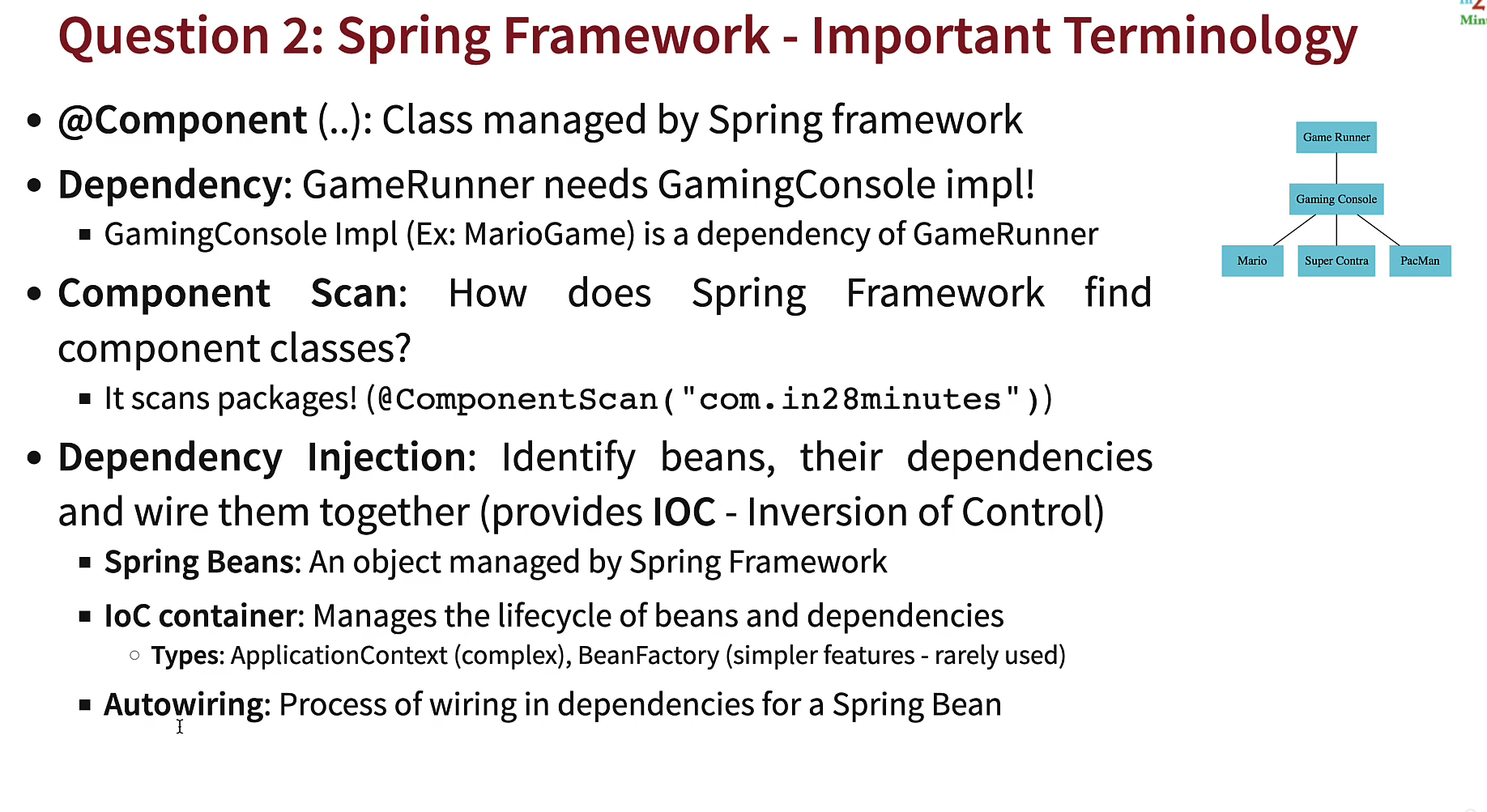
**Challenges with Java Projects Before Spring:**

* Boilerplate Code: A significant amount of repetitive code (e.g., for JDBC, transaction management) had to be written.
* Tight Coupling: Classes were tightly coupled, making applications less flexible and harder to test or extend.
* Complex Configuration: Managing configurations (e.g., in web.xml) was tedious and error-prone.
* Difficulty in Testing: Testing Java EE applications required complex setups, such as deploying the application to a server.
* Heavyweight Containers: Enterprise Java Beans (EJBs) were heavy and required specific application servers.
* Limited Modularization: It was hard to modularize and reuse application components effectively.

Spring was designed to address these issues by providing a lightweight, flexible, and easy-to-use alternative to traditional Java EE.

Key concepts of Spring:

1. Dependency Injection
2. Inversion of control (IOC)



## 1.1 Inversion of Control(IoC)

IoC is a design principle where the control of object creation and their dependencies is transferred from the application code to the Spring IoC container.

**How It Works:**

* In traditional Java, you create and manage dependencies manually (e.g., using new).
* With IoC, the Spring container creates objects (beans), manages their lifecycle, and injects dependencies where needed.

**Benefits:**

* Reduces tight coupling between classes.
* Simplifies testing and promotes better design.

**Key Concepts:**

* **Component:** Class managed by Spring framework.
* **IoC Container:** Manages the lifecycle and configuration of application objects.
* **Bean:** An object managed by the IoC container.
* **Dependency Injection (DI):** A way to provide dependencies to objects managed by the container.

@Component  
public class Engine {}  
  
@Component  
public class Car {  
 private final Engine engine;  
  
 @Autowired  
 public Car(Engine engine) {  
 this.engine = engine; // Dependency injected by Spring  
 }  
}

## 1.2 Dependency Injection

DI is a specific implementation of IoC, where dependencies (objects or services) are provided to a class by the Spring container rather than being instantiated by the class itself.

**Types of DI:**

* Constructor Injection: Dependencies are passed via constructor.
* Setter Injection: Dependencies are injected using setter methods.
* Field Injection: Dependencies are injected directly into fields (using annotations like @Autowired).

**Benefits:**

* Promotes loose coupling.
* Simplifies testing and code reusability.

Aspect-Oriented Programming (AOP)

AOP is a programming paradigm that allows you to modularize cross-cutting concerns (like logging, security, and transactions) by separating them from the main business logic.

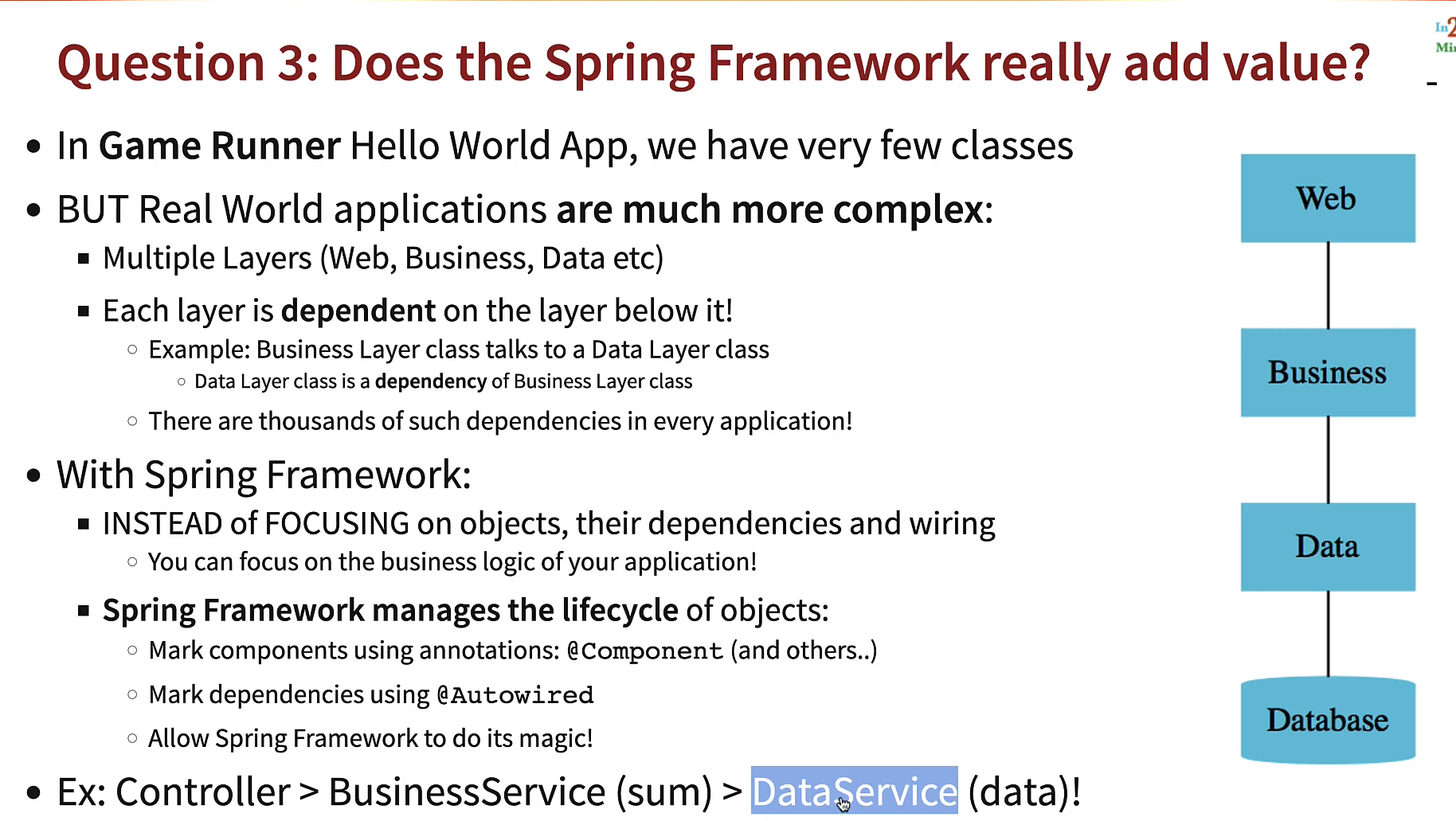
**Key Terminologies:**

* **Aspect:** A module that encapsulates cross-cutting concerns.
* **Advice:** The action performed by an aspect (e.g., logging).
* **Join Point:** A point in the application (e.g., method execution) where an advice can be applied.
* **Pointcut:** A set of join points where an advice is executed.

**Benefits:**

* Eliminates repetitive code.
* Centralizes cross-cutting concerns.

@Aspect  
@Component  
public class LoggingAspect {  
 @Before("execution(\* com.example.service.\*.\*(..))")  
 public void logBeforeMethodExecution() {  
 System.*out*.println("Method is about to be executed.");  
 }  
}



Eg:- Data coming from Data Layer, Business Service will handle business logic and controller class handles user interaction and manages HTTP requests.

import java.util.List;  
  
@Component  
public class DataService {  
  
 public List getData(){  
 List<Integer> nums = Arrays.*asList*(10,20,30,40);  
 return nums;  
 }  
}

package com.practice.LearnSpring.rest;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Component;  
import java.util.List;  
  
@Component  
public class BusinessService {  
  
 @Autowired  
 private DataService dataService;  
 public long calculateSum(){  
 List<Integer> data = dataService.getData();  
 long total = data.stream().reduce(Integer::*sum*).get();  
 return total;  
 }  
}

package com.practice.LearnSpring.rest;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Component;  
  
@Component  
public class WebController {  
 @Autowired  
 private BusinessService businessService;  
 public long getBusinessServiceValue(){  
 return businessService.calculateSum();  
 }  
}

package com.practice.LearnSpring;  
import com.practice.LearnSpring.rest.WebController;  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.context.ConfigurableApplicationContext;  
  
@SpringBootApplication  
public class LearnSpringApplication {  
  
 public static void main(String[] args) {  
 ConfigurableApplicationContext context = SpringApplication.*run*(LearnSpringApplication.class, args);  
 WebController controller = context.getBean(WebController.class);  
 System.*out*.println(controller.getBusinessServiceValue());  
 }  
  
}

* **Controller:** Receives the user request and passes it to the Business Layer.
* **Business Layer:** Processes the request, applies logic, and interacts with the Data Layer.
* **Data Layer:** Fetches or stores data in the database and returns the result to the Business Layer.

## 1.3 Types of Dependency Injection

Dependency Injection (DI) is a design pattern that allows the Spring IoC container to inject dependencies into an object. The three primary types of DI in Spring are Constructor Injection, Setter Injection, and Field Injection. Let’s explore each with examples.

1. Constructor Injection

The dependency is provided through the class constructor.

Advantages:

* Dependencies are immutable (final fields can be used).
* Ensures mandatory dependencies are provided.

Disadvantages:

* Becomes verbose if there are many dependencies.

@Component  
public class Engine {  
 public void start() {  
 System.*out*.println("Engine started.");  
 }  
}  
  
@Component  
public class Car {  
 private final Engine engine;  
  
 @Autowired // Optional in modern Spring versions  
 public Car(Engine engine) {  
 this.engine = engine; // Dependency injected here  
 }  
  
 public void drive() {  
 engine.start();  
 System.*out*.println("Car is driving.");  
 }  
}  
  
@SpringBootApplication  
public class App {  
 public static void main(String[] args) {  
 ApplicationContext context = SpringApplication.*run*(App.class, args);  
 Car car = context.getBean(Car.class);  
 car.drive();  
 }  
}

1. Setter Injection

The dependency is provided through a public setter method.

Advantages:

* Useful for optional dependencies.
* More readable when there are multiple dependencies.

Disadvantages:

* Dependencies can be modified after the object is initialized, making it less secure.

@Component  
public class Engine {  
 public void start() {  
 System.*out*.println("Engine started.");  
 }  
}  
  
@Component  
public class Car {  
 private Engine engine;  
  
 @Autowired  
 public void setEngine(Engine engine) {  
 this.engine = engine; // Dependency injected here  
 }  
  
 public void drive() {  
 engine.start();  
 System.*out*.println("Car is driving.");  
 }  
}  
  
@SpringBootApplication  
public class App {  
 public static void main(String[] args) {  
 ApplicationContext context = SpringApplication.*run*(App.class, args);  
 Car car = context.getBean(Car.class);  
 car.drive();  
 }  
}

1. Field Injection

The dependency is directly injected into a field using the @Autowired annotation.

Advantages:

* Simplest to implement and read.
* Requires minimal boilerplate code.

Disadvantages:

* Not suitable for mandatory dependencies (hard to verify during testing).
* Makes the class less flexible (dependency cannot be replaced dynamically).

@Component  
public class Engine {  
 public void start() {  
 System.*out*.println("Engine started.");  
 }  
}  
  
@Component  
public class Car {  
 @Autowired  
 private Engine engine; // Dependency injected here  
  
 public void drive() {  
 engine.start();  
 System.*out*.println("Car is driving.");  
 }  
}  
  
@SpringBootApplication  
public class App {  
 public static void main(String[] args) {  
 ApplicationContext context = SpringApplication.*run*(App.class, args);  
 Car car = context.getBean(Car.class);  
 car.drive();  
 }  
}

**Comparison of Dependency Injection Types**

| **Feature** | **Constructor Injection** | **Setter Injection** | **Field Injection** |
| --- | --- | --- | --- |
| **Mandatory Dependencies** | Yes | No | No |
| **Immutability** | Yes | No | No |
| **Ease of Testing** | High | Medium | Low |
| **Code Simplicity** | Medium | High | High |
| **Best Use Case** | Mandatory and immutable dependencies | Optional dependencies | Simple use cases where testing flexibility is not required |

**Conclusion**

* **Use Constructor Injection** for mandatory, immutable dependencies.
* **Use Setter Injection** for optional or dynamically configurable dependencies.
* **Use Field Injection** for simple cases, but avoid it in complex scenarios or when testing flexibility is essential.

Spring suggests constructor based injection

4. Key Components of Spring

1. Spring Core

Provides the core features like IoC and DI.

2. Spring AOP

Enables aspect-oriented programming for separating cross-cutting concerns.

3. Spring Data

Simplifies data access and database operations.

4. Spring MVC

Provides a framework for building web applications.

5. Spring Security

Handles authentication and authorization.

6. Spring Boot

Simplifies Spring-based application development by providing auto-configuration, embedded servers, and production-ready features.

7. Spring Cloud

Helps build microservices with features like service discovery, configuration management, and distributed tracing.

## Types of Annotations

| **Annotation** | **Purpose** |
| --- | --- |
| @Component | Marks a generic Spring-managed bean. |
| @Controller | Handles HTTP requests in MVC apps(more specific type of component, class level annotation) |
| @Service | Defines business logic components(more specific type of component, class level annotation) |
| @Repository | Handles database access components(more specific type of component, class level annotation) |
| @Autowired | Injects dependencies automatically(creates lose coupling) |
| @Configuration | Defines configuration classes. |
| @Bean | Declares beans in configuration classes. |
| @Qualifier | Resolves dependency injection ambiguity. |
| @Scope | Defines the scope of a bean. |
| @Transactional | Manages transactions. |

@value

# **What is Spring Boot?**

Spring Boot is an open-source framework built on top of the Spring Framework. It simplifies the process of creating Spring-based applications by eliminating the need for extensive boilerplate configuration. With opinionated defaults, embedded servers, and a focus on rapid development, Spring Boot enables developers to quickly build production-ready applications.

## 2.1. Key Features of Spring boot

1. **Auto-Configuration:**
   * Automatically configures Spring components based on project dependencies.
   * Example: If spring-boot-starter-web is included, Spring Boot configures a web server.
2. **Starter Dependencies:**
   * Simplifies dependency management with pre-defined groups of dependencies.
   * Example: spring-boot-starter-data-jpa includes JPA, Hibernate, and database drivers.
3. **Embedded Servers:**
   * Eliminates the need for external application servers.
   * Supports Tomcat, Jetty, and Undertow.
4. **Spring Initializr:**
   * A web-based tool to bootstrap Spring Boot projects with minimal setup.
   * Accessible at [start.spring.io](https://start.spring.io).
5. **Production-Ready Features:**
   * Provides **Actuator** for monitoring, health checks, metrics, and application insights.
6. **Command-Line Interface (CLI):**
   * Allows rapid development using Groovy scripts.
7. **Microservices Support:**
   * Ideal for building microservices with features like embedded servers and lightweight configurations.
8. **Externalized Configuration:**
   * Supports configuration via application.properties or application.yml files.
9. **Security Integration:**
   * Simplifies adding authentication and authorization using **Spring Security**.
10. **Test Support:**
    * Provides testing tools with pre-configured annotations like @SpringBootTest.

### **How is Spring Boot Different from Spring?**

| **Feature** | **Spring** | **Spring Boot** |
| --- | --- | --- |
| **Setup and Configuration** | Requires manual XML or Java-based configuration. | Provides auto-configuration and starter dependencies. |
| **Application Server** | Requires external servers like Tomcat, Jetty, etc. | Comes with embedded servers (e.g., Tomcat, Jetty, Undertow). |
| **Dependency Management** | Dependencies are manually configured. | Provides starter dependencies (e.g., spring-boot-starter-web). |
| **Project Initialization** | Manual setup with various configurations. | Simplified project setup with **Spring Initializr**. |
| **Production-Ready Features** | Requires custom configurations. | Includes monitoring, metrics, and health checks by default. |
| **Ease of Use** | Requires more effort for setup and development. | Designed for quick, out-of-the-box development. |

Example 1:

Course.java

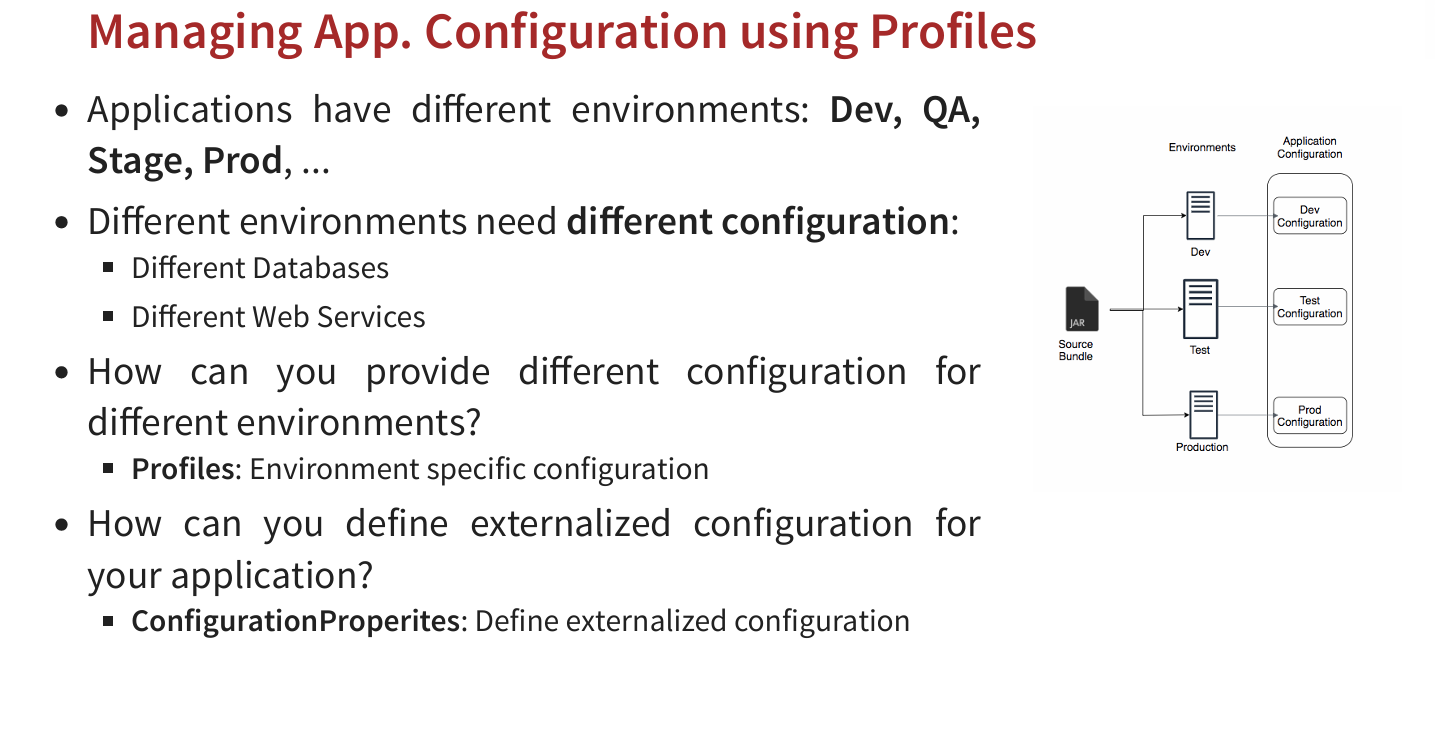
package com.practice.LearnSpring.Course;  
  
public class Course {  
 int id;  
 String name;  
  
 public Course(int id, String name) {  
 this.id = id;  
 this.name = name;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 @Override  
 public String toString() {  
 return "Course{" +  
 "id=" + id +  
 ", name='" + name + '\'' +  
 '}';  
 }  
  
 public int getId() {  
 return id;  
 }  
  
 public void setId(int id) {  
 this.id = id;  
 }  
}

CourseController

package com.practice.LearnSpring.Course;  
  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.RestController;  
import java.util.ArrayList;  
import java.util.List;  
  
@RestController  
public class CourseController {  
 @RequestMapping("/course")  
 public List<Course> getCourses(){  
 List<Course> courses = new ArrayList<>();  
 courses.add(new Course(1,"Maths"));  
 courses.add(new Course(2,"English"));  
 return courses;  
 }  
}

Servelets > Spring MVC > spring boot(from downloaded video)

Profiles:



## 2.2. Configuration Properties:

Lets say we want to use configurations of application.properties in our code

So in application.properties

Temp-service.url = "https://palash-bajpai-45.netlify.app/"  
temp-service.username="Palash"  
temp-service.password="abcdefg"

Create a class, with @ConfigurationProperties annotation

package com.practice.LearnSpring.learnConfiguration;  
  
import org.springframework.boot.context.properties.ConfigurationProperties;  
import org.springframework.stereotype.Component;  
  
@ConfigurationProperties(prefix = "temp-service")  
@Component  
public class TempConfig {  
 private String url;  
 private String username;  
 private String password;  
  
 public String getUsername() {  
 return username;  
 }  
  
 public void setUsername(String username) {  
 this.username = username;  
 }  
  
 public String getUrl() {  
 return url;  
 }  
  
 public void setUrl(String url) {  
 this.url = url;  
 }  
  
 public String getPassword() {  
 return password;  
 }  
  
 public void setPassword(String password) {  
 this.password = password;  
 }  
  
  
}

Can create a rest controller to use it

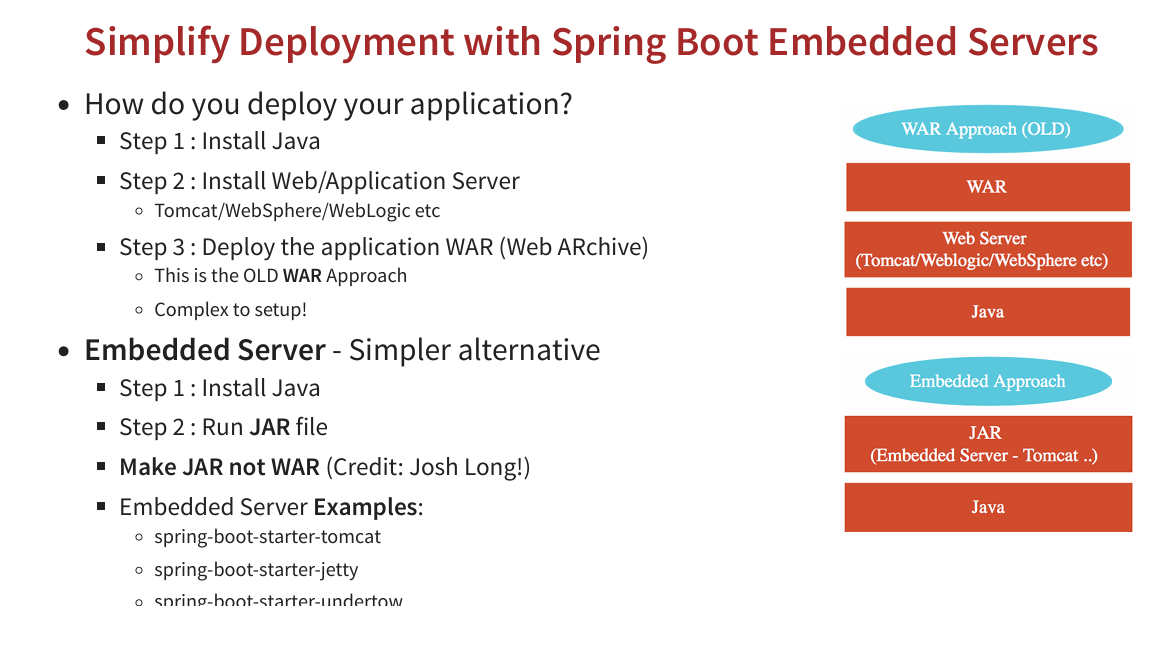
package com.practice.LearnSpring.learnConfiguration;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.RestController;  
  
@RestController  
public class TempController {  
  
 @Autowired  
 private TempConfig tempConfig;  
  
 @RequestMapping("/tempConfig")  
 public TempConfig getTempConfig(){  
 return tempConfig;  
 }  
}

## Embedded Servers

War VS Jar. How spring bot improved

How jar is better

Maven clean install : create jar , try to launch jar, java –jar jarname



## Actuators

By default it only shows health in localhost:8080/actuator. To get more information

In application.properties: add

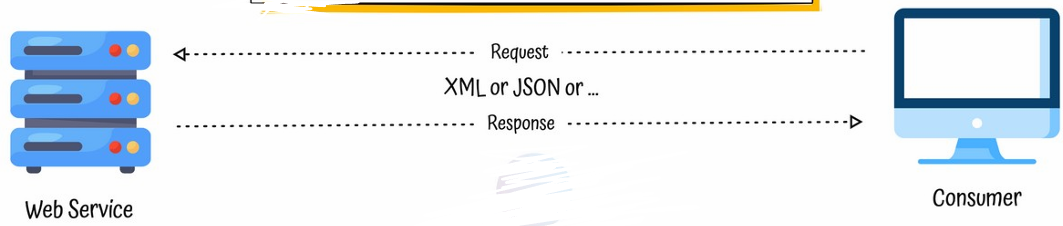
management.endpoints.web.exposure.include=\*

# **3. Web Service**

A web service is a standardized way of enabling communication and data exchange between two software applications over a network (typically the Internet). These applications can be written in different programming languages and run on different platforms. Web services use protocols and standards to facilitate interoperability.

**Key Characteristics:**

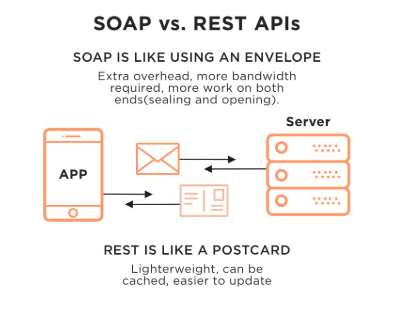
* **Platform-Independent:** Web services enable applications written in different languages (e.g., Java, Python, .NET) to communicate seamlessly. Also communication should be platform-independent (eg XML, Json).
* **Protocol-Based Communication:** Allows communication over a network and uses protocols like HTTP, HTTPS, and SOAP for exchanging data.
* **Loosely Coupled:** The client and server are not tightly bound; they communicate through interfaces defined by the service.
* **Interoperability:** Web services ensure interoperability across systems with different architectures.



**Key Components:**

* **Request:** Input to a web service
* **Response:** Output from the web service
* **Message Exchange Format (Payload Format):** XML or JSON for exchanging data.
* **Endpoints:** URL where the web services are accessible.
* **Service Provider (Server):** Entity providing the Web Service.
* **Service Consumer:** Entity consuming the Web Service.

## 3.1 Types of web service



**1. SOAP (Simple Object Access Protocol) Web Services**

**Definition:** SOAP is a protocol for exchanging structured information using XML over HTTP, SMTP, or other transport protocols.

**Features:**

Strict standards for message format.

Built-in error handling.

Provides higher security (WS-Security).

**Advantages:**

Ideal for enterprise-level applications requiring reliability.

Supports multiple transport protocols.

**Disadvantages:**

Verbose and complex to implement.

Slower due to XML-based payload.

**Example:** A banking service providing secure account details.

**2. REST (Representational State Transfer) Web Services**

**Definition:** REST is an architectural style for designing lightweight web services using standard HTTP methods.

**Features:**

Stateless communication.

Supports multiple formats (XML, JSON, etc.).

Simpler to implement compared to SOAP.

**Advantages:**

Lightweight and faster.

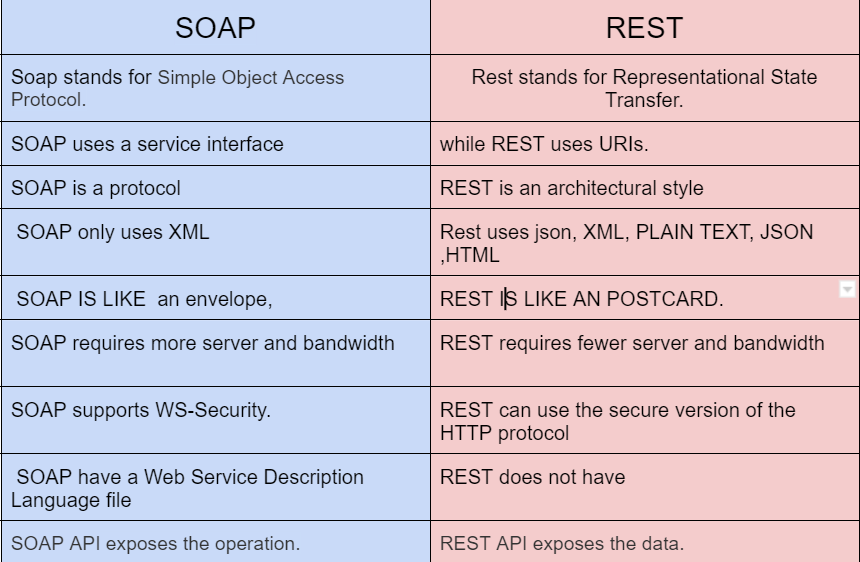
Easier to integrate with modern web applications.

Broadly used for APIs.

**Disadvantages:**

Lacks built-in security mechanisms (depends on HTTPS).

**Example:** A REST API for retrieving weather data.



# **4. REST API with Spring Boot**

Create application > spring initializer

Add dependencies: spring web: for Rest APi,

Spring data JPA: to store data in SQL

H2 database: in-memory database

Spring boot Dev tools: provide Live Reload.

Basic Hello world

@RestController  
public class HelloWorldController {  
 @RequestMapping(method = RequestMethod.*GET*, path="/hello")  
 public String hello(){  
 return "Hello World";  
 }  
}

Can directly use

@RestController  
public class HelloWorldController {  
 @GetMapping("/hello")  
 public String hello(){  
 return "Hello World";  
 }  
}

## 4.1. Security

# **5. Intro to Microservice**

Microservices, also known as the Microservices architecture, is a software development approach where a large application is divided into smaller, loosely coupled, and independently deployable services. Each service performs a specific function and communicates with other services using lightweight protocols.

**Core Concepts:**

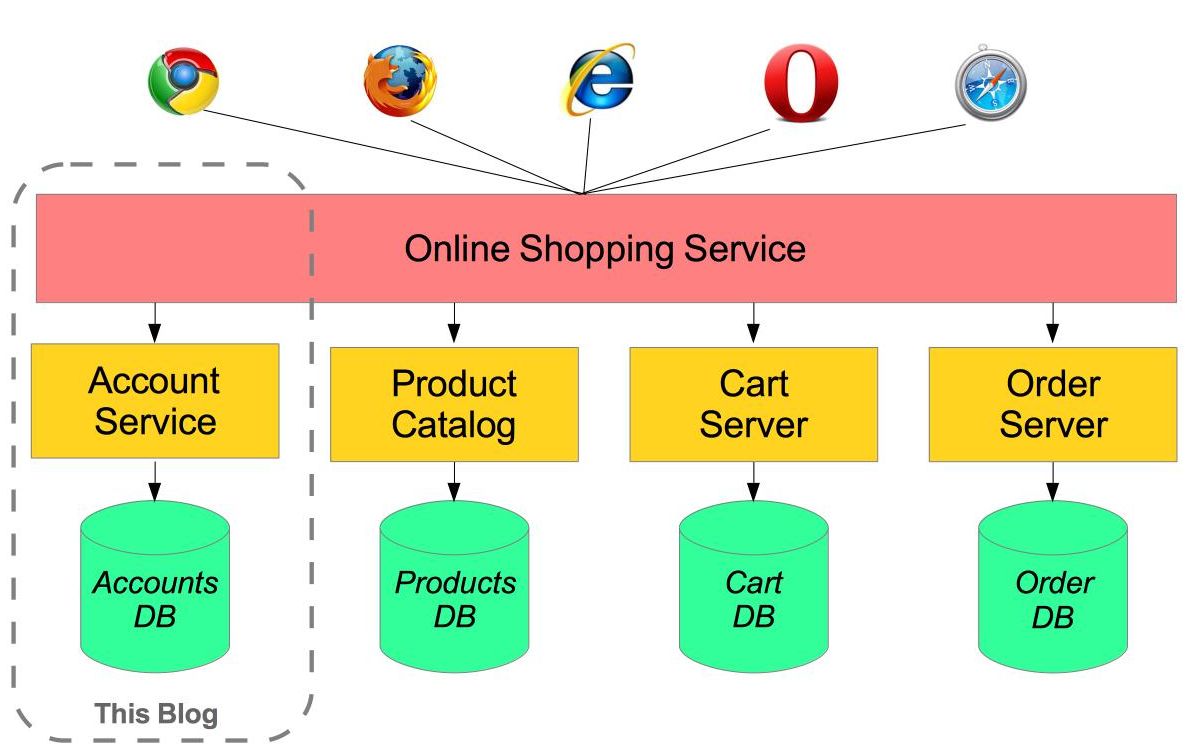
* **Decentralization:** Microservices decentralize development, allowing teams to work independently on different services.
* **Service Ownership:** Each microservice is owned by a specific team, ensuring accountability and focus.
* **Technology Agnostic:** Each service can use different technologies, programming languages, or databases.
* **Single Responsibility:** Each service is responsible for a specific task or business domain.

**Key Features:** Independent deployment and testing, Scalability (horizontal scaling), Modularity, Loose Coupling, Fault Isolation, and Technology Diversity and easier integration with modern technologies (DevOps, cloud-native).

**Disadvantages:** Increased complexity and delay in inter-communication, Dependency on robust monitoring and logging tools.

**Real world example:**

E-commerce platform: Amazon uses microservices to manage its vast ecosystem of functionalities like product search, recommendations, user authentication, and payments.

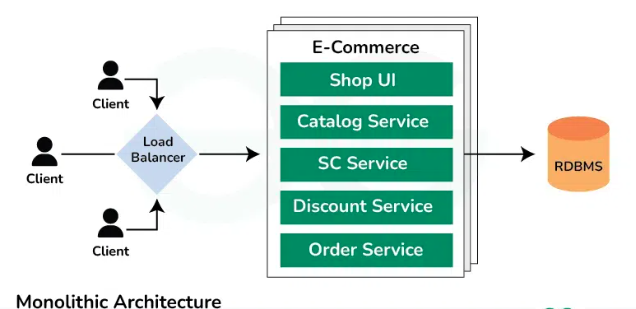


## 5.1. Monolithic Architecture

Monolithic architecture is a traditional software development approach where all components of an application—such as the user interface, business logic, and data layer—are tightly integrated into a single, unified codebase.

**Core Concepts:**

* **Unified Codebase:** All modules (e.g., authentication, payment, reporting) are part of one application.
* **Single Deployment Unit:** The entire application is deployed as a single unit on servers.
* **Centralized Data Management**: Typically uses a single database for all features and services.
* **Tightly Coupled Components:** Changes in one module often affect others, requiring comprehensive testing.



**Advantages: -** Faster(inter module communication not required) , Easier Deployment lifecycle), ideal for smaller project or project which don’t need high scalability.

**Disadvantages: -** Scalability challenges, harder to debug, difficulty in maintenance and updates, issue in one place bring down entire system, lack of flexibility for modern needs.

To overcome all this issues, we use microservice architecture.

# **4. Interview Questions**

**3. How Web Services Solve the Issue of Sharing Code?**

Problems with Sharing Code Directly:

* **Compatibility Issues:** Sharing a JAR file restricts usage to specific programming languages and platforms.
* **Distribution Challenges:** Every time the code changes, the updated JAR must be redistributed.
* **Versioning Problems:** Managing multiple versions of the same library is complex.
* **Limited Accessibility:** Sharing a JAR file does not support real-time or remote access.

Solution via Web Services:

* **Real-Time Access:** The client consumes the service in real-time, ensuring the latest version of the code is used.
* **Language Agnosticism:** Web services provide data in standardized formats like XML or JSON, enabling usage across languages.
* **Centralized Updates:** Updates to the service are made on the server, automatically reflecting for all clients.
* **Ease of Integration:** With standard protocols like HTTP and REST, integration with web services is straightforward.

### **Why Use Spring Boot?**

* **Faster Development:** Opinionated defaults and auto-configuration speed up development.
* **Microservices-Friendly:** Embedded servers and lightweight nature make it ideal for microservices.
* **Production Ready:** Integrated features like Actuator for monitoring and metrics.
* **Simplified Testing:** Pre-configured testing environment reduces complexity.
* **Ease of Configuration:** Externalized configuration supports flexibility.

### **Difference Between** @Controller **and** @RestController

| **Aspect** | **@Controller** | **@RestController** |
| --- | --- | --- |
| **Purpose** | Used for handling web requests and returning views (e.g., HTML, JSP). | Used for building REST APIs and returning data (JSON/XML). |
| **Annotation** | Typically used with @ResponseBody for returning data. | Combines @Controller and @ResponseBody. |
| **Response Type** | Returns a view name that resolves to a template. | Returns data directly as the HTTP response body. |
| **Use Case** | Ideal for MVC applications. | Ideal for RESTful APIs. |

### **Examples**

#### **1. Using** @Controller

The @Controller annotation requires an additional @ResponseBody annotation if the response is JSON or XML.

java

CopyEdit

@Controller

@RequestMapping("/api")

public class UserController {

@GetMapping("/user")

@ResponseBody // Explicitly marks this method to return data as a response body.

public User getUser() {

return new User("John", "Doe"); // Returns JSON/XML representation of the User object.

}

@GetMapping("/home")

public String homePage() {

return "home"; // Resolves to a view named "home.html" or "home.jsp".

}

}

#### **2. Using** @RestController

The @RestController simplifies the creation of REST APIs by implicitly applying @ResponseBody to all methods.

java

CopyEdit

@RestController

@RequestMapping("/api")

public class UserRestController {

@GetMapping("/user")

public User getUser() {

return new User("John", "Doe"); // Automatically returns JSON/XML without needing @ResponseBody.

}

@GetMapping("/greeting")

public String greeting() {

return "Hello, World!"; // Directly returns the string as a response.

}

}