Spring Core

SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://softuni.bg

Questions?

Software University

sli.do

#java-web

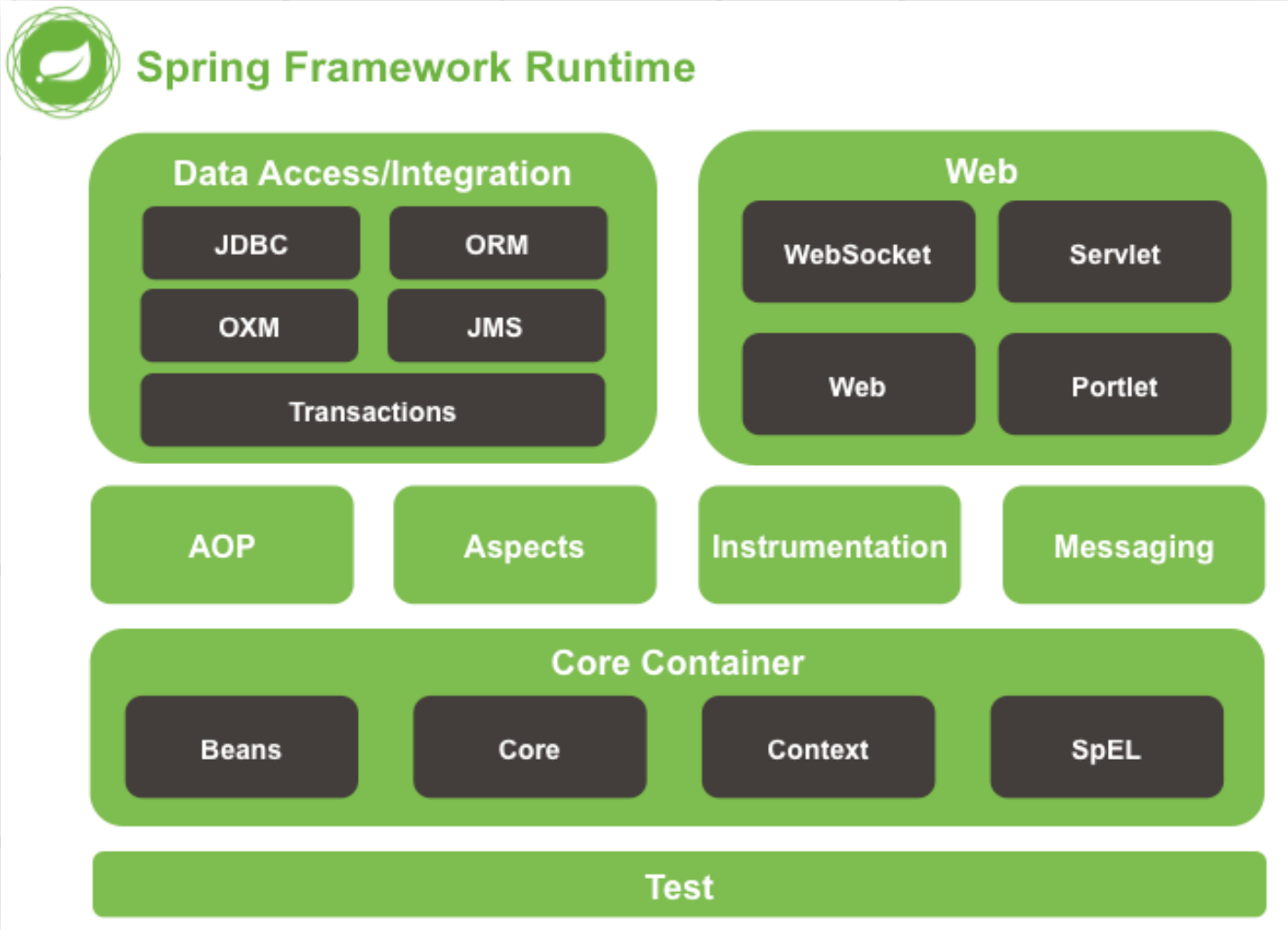
Table of Contents

What is Spring?

What is Spring?

- Spring is a powerful, **open-source framework** that simplifies Java application development
- **Main Benefits**:
  - Brings tools like **Spring MVC** (web), **Spring Data** (database), etc.
  - Simplifies dependency management between objects (**DI**)
  - Manages object creation and lifecycle (**IoC**)
  - Handles **cross-cutting concerns** (e.g. security)

What is Spring?

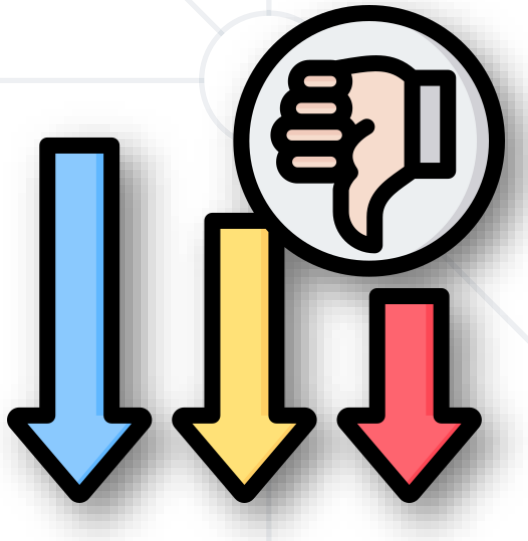


Spring Framework Core Features

- **Inversion of Control (IoC)**:
  - Traditional programming leading to **tight coupling**
  - **IoC** delegates **object creation** to IoC Container
  - **IoC** promotes **loose coupling**
  - This design is fundamental to the **Spring Framework**
- **Dependency Injection (DI)**:
  - A **technique** where objects **receive dependencies** externally
  - **Eliminates the need** for objects **to create** their own dependencies

Spring Disadvantages

- Complex Configuration
- Manual Module Management
- Manual Setup and Configuration
- Long Startup Times



What is Spring Boot?

What is Spring Boot?

- **Spring Boot** is a Spring extension designed for rapid application development
- Offering tools for **faster** and **easier** configuration on top of the Spring Framework

Spring Boot Concepts

- **Opinionated Defaults**:
  - Spring Boot **makes decisions** for you, reducing setup time
  - Example: **spring-boot-starter-web** automatically configures **Tomcat** and Spring **MVC**
- **Starter Dependencies**:
  - Bundles commonly used libraries into simplified dependencies
  - Example: **spring-boot-starter-jpa** includes Hibernate and Spring Data JPA

Spring Boot Concepts

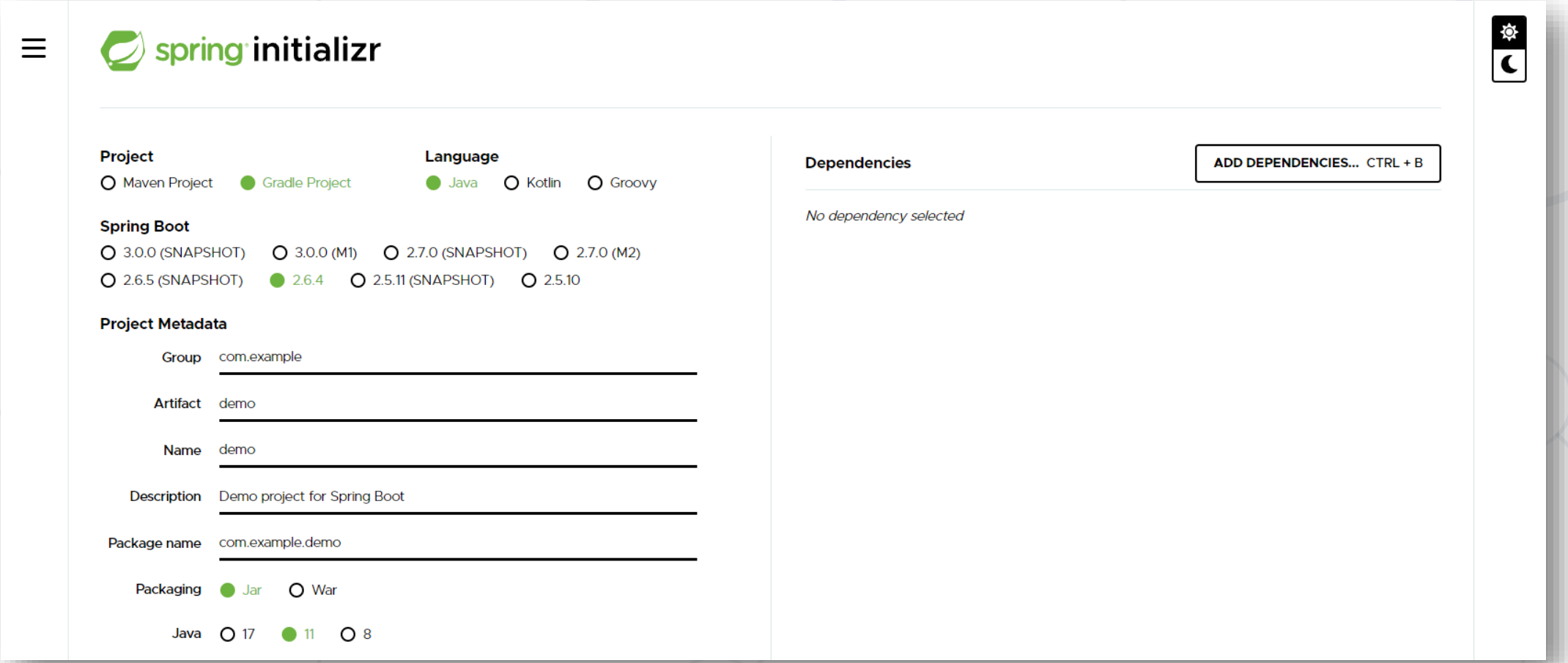
- **Production-Ready Features**:
  - Tools like **Spring Boot Actuator** provide metrics, health checks, and management capabilities for production

Spring Boot Concepts

- **Spring Framework**:
  - Foundational building blocks
  - Backbone of the Spring ecosystem
- **Spring Boot**:
  - Adds auto-configuration, embedded servers, and starters packages
  - Designed for rapid development

Creating Spring Boot Project

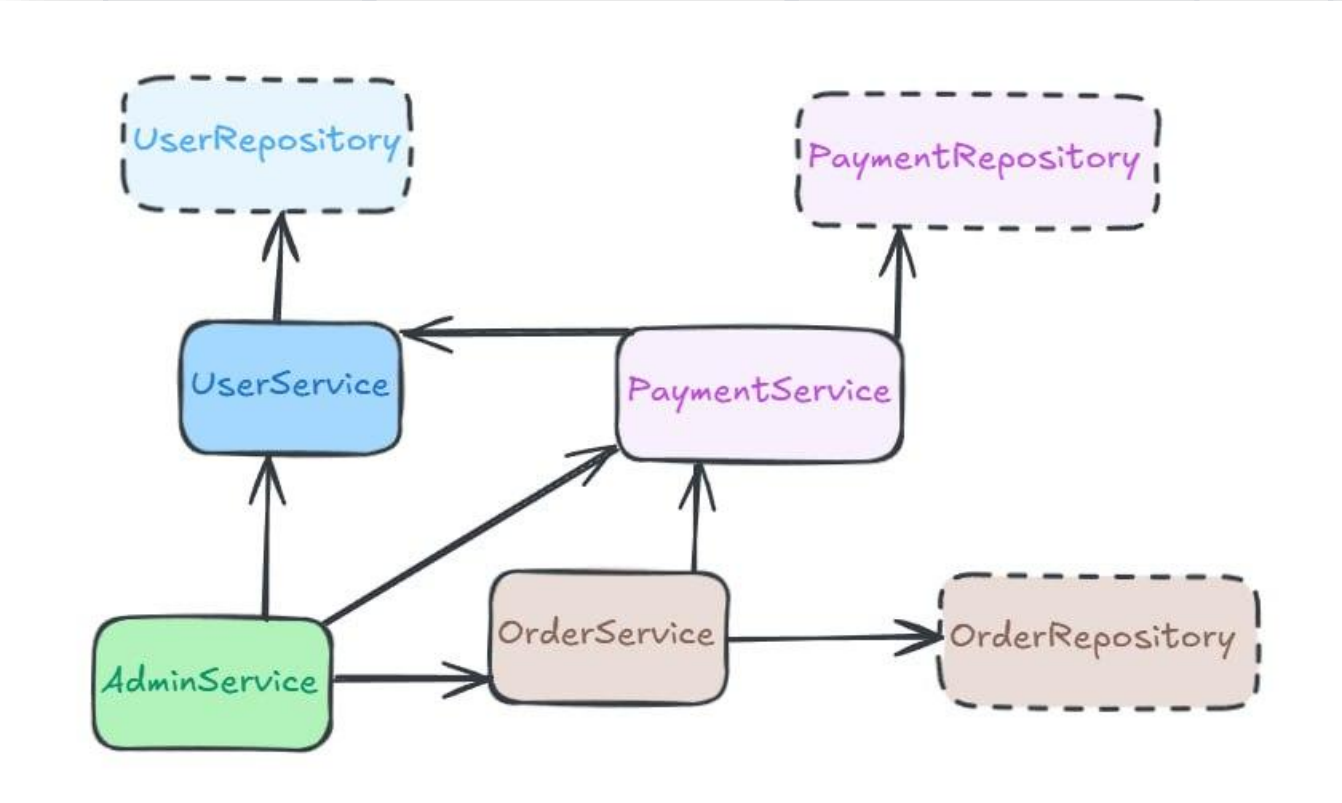
- Just go to **https://start.spring.io/**



Inversion of Control (IoC)

Object Network

- An application contains many components that **relate/depend** on each other



Tight Coupling

- An application needs a **single instance** of a business object

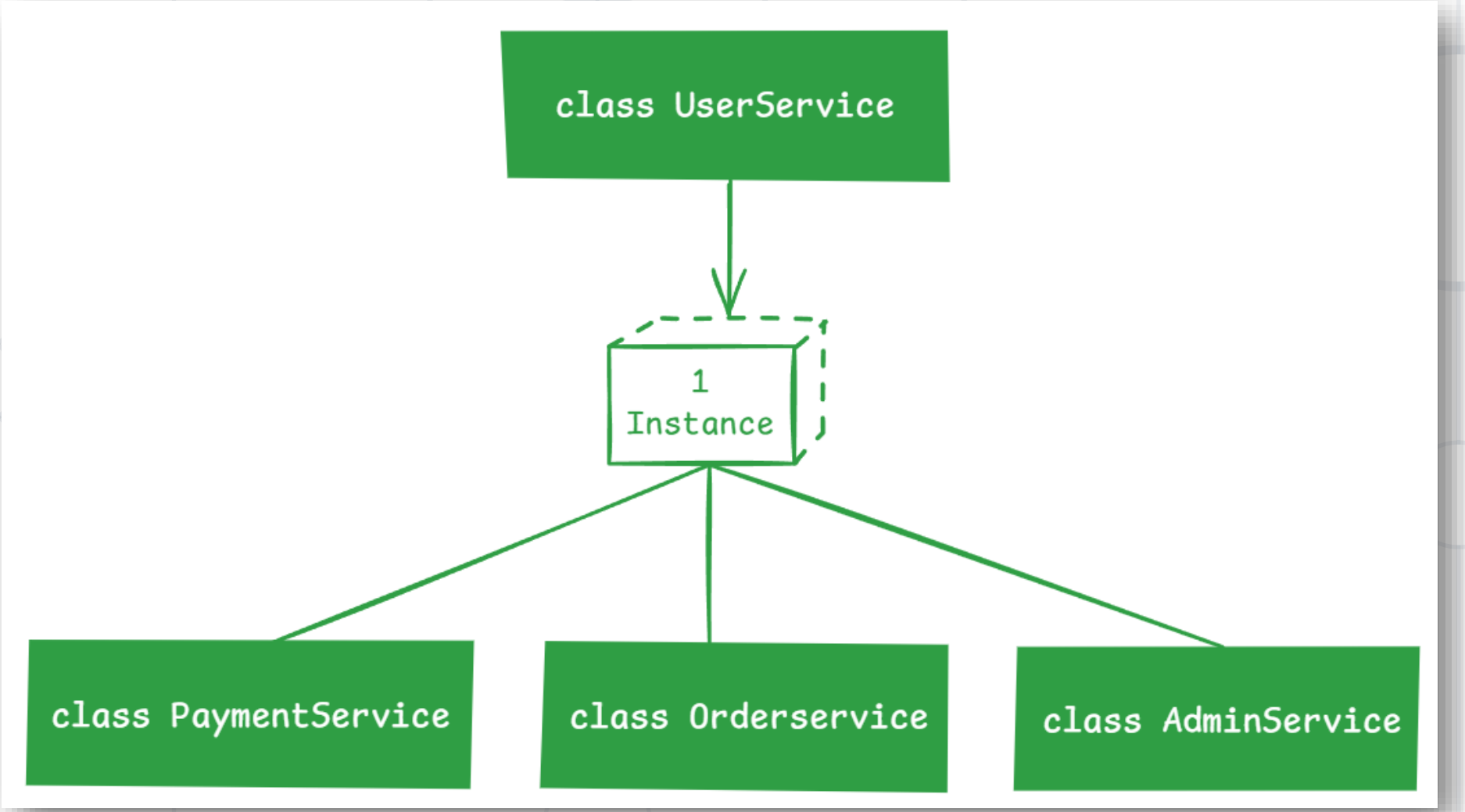
3 instances of the same class created at runtime due to tight coupling

```
public class AdminService {

    private final UserService userService;

    public AdminService() {
        this.userService = new UserService();
    }                                  +1 instance

    public void checkUserActivity() {...}
}
```

```
public class PaymentService {

    private final UserService userService;

    public PaymentService() {
        this.userService = new UserService();
    }                                  +1 instance

    public void makePayment() {...}
}
```

```
public class OrderService {

    private final UserService userService;

    public OrderService() {
        this.userService = new UserService();
    }                                  +1 instance

    public void sendOrder() {...}
}
```

Shared Instance



Inversion of Control (IoC)

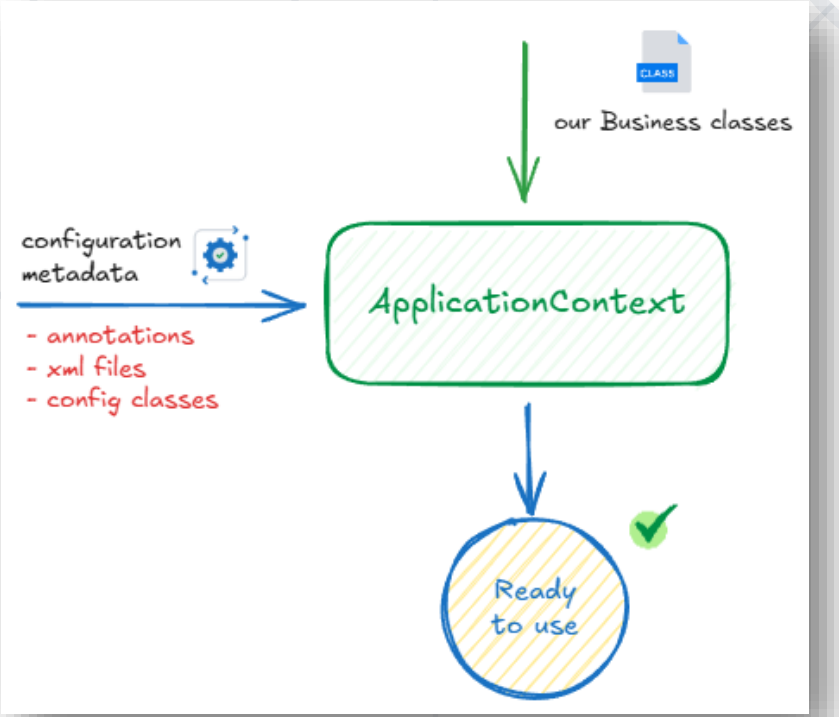
- **Design principle** where the **control** of object creation and lifecycle is **delegated to** a framework/container, instead of the developer
- IoC Container:
  - The core **component** that realizes the IoC principle
  - It handles object creation, lifecycle management, and dependencies, leading to loose coupling between objects

What Can IoC Containers Do?

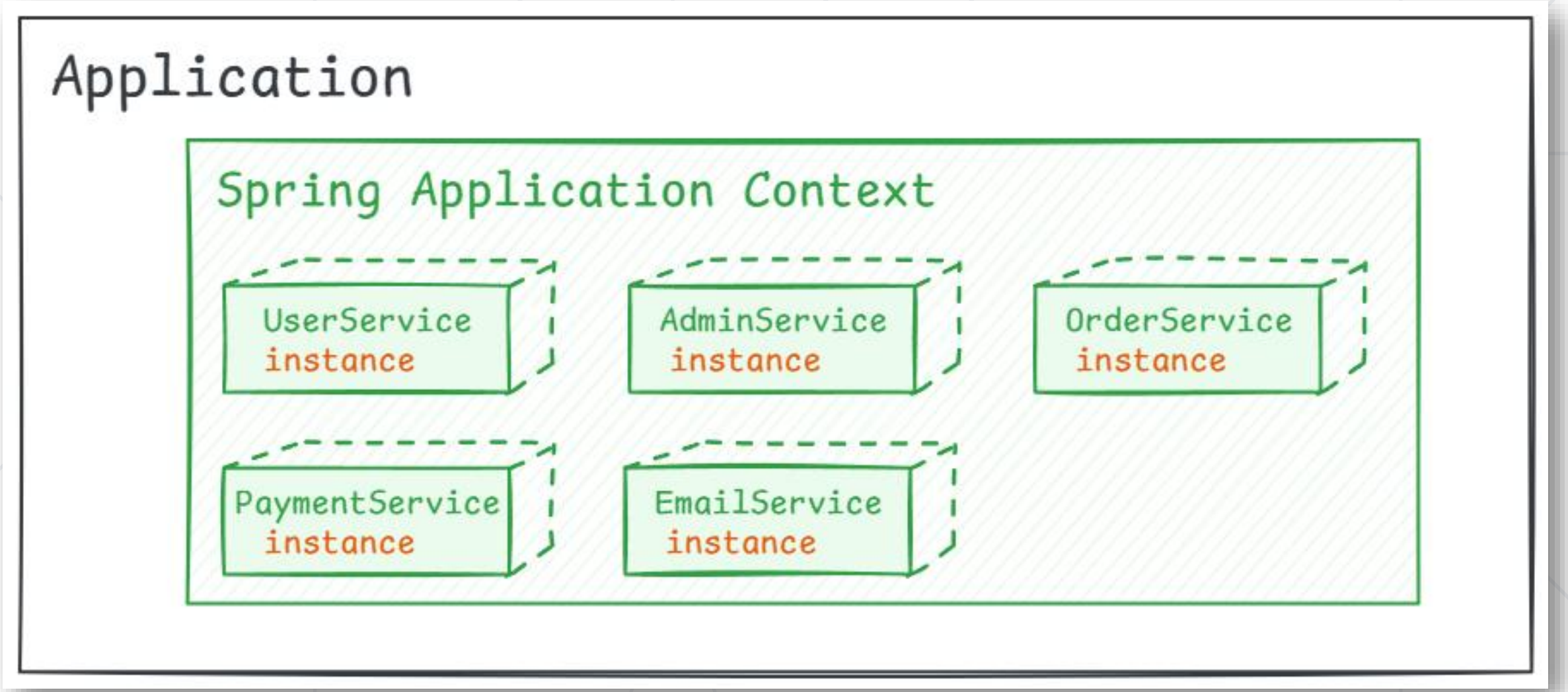
- **Object Creation**: Instantiate objects as defined in the configuration
- **Dependency Injection**: Inject required dependencies
- **Lifecycle Management**: Initialize and destroy objects

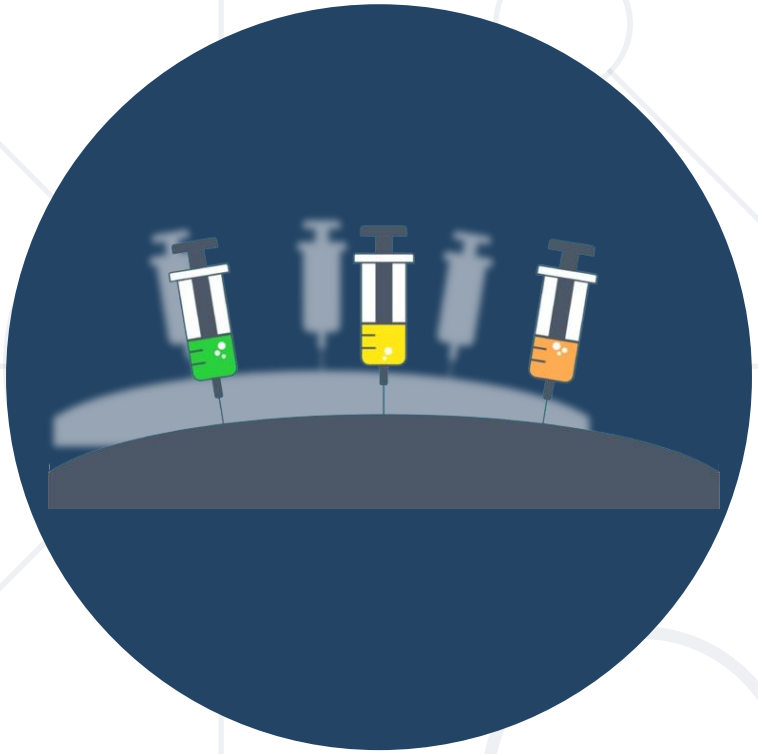
ApplicationContext – Spring's IoC Container

- The main **IoC container** in **Spring**
- Manages class **instances / objects** (beans)
- Provide **dependencies** exactly where they are required



Spring Application Context





Dependency Injection (DI)

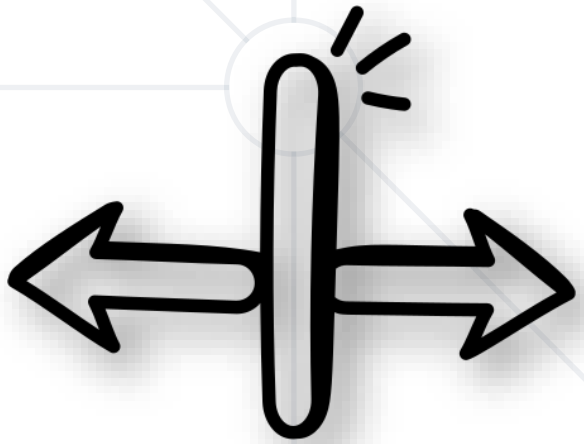
What is Dependency Injection

- **Dependency Injection** is a technique in Spring where the framework provides the required dependencies to a class, instead of the class creating them itself
- DI promotes **Separation of Concerns**, allowing developers to focus on the class's logic without worrying about how its dependencies are created or managed

Separation of Concerns

- **Eliminates** Responsibility for Object Creation

```
// Not correct
public class OrderService {
    private ProductService productService = new ProductService();
}
```



Separation of Concerns

- With DI, components (Java classes) can be developed and tested **independently** of their dependencies (their fields)

```
// Correct
@Service
public class OrderService {
    private final ProductService productService;
    @Autowired
    public OrderService(ProductService productService) {
        this.productService = productService;
    }
}
```

Types of DI

- **Constructor Injection**: Dependencies are injected via **constructor** parameters
  - This is the most common and recommended form of DI

```
@Service
public class OrderService {
    private final ProductService productService;
    @Autowired
    public OrderService(ProductService productService) {
        this.productService = productService;
    }

    // some methods
}
```

Types of DI

- **Setter Injection**: Dependencies are injected via setter methods on the **dependent** class

```
@Service
public class OrderService {
    private ProductService productService;
    @Autowired
    public void setProductService(ProductService productService) {
        this.productService = productService;
    }
    // some methods
}
```

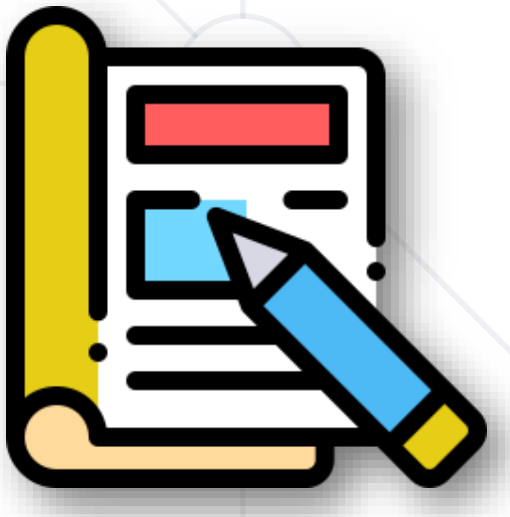
Types of DI

- **Field Injection**: Dependencies are injected **directly** into fields of the dependent class
  - This approach is **less** preferred due to its potential drawbacks, such as reduced testability and tight coupling

```
@Service
public class OrderService {
    @Autowired
    private ProductService productService;


    // some methods
}
```

@Autowired Annotation

- Used in Spring to **automatically inject dependencies** into a class by the IoC container
- Can be applied to **fields**, **constructors**, or **setter methods**
- The **IoC** container resolves and provides the required bean at **runtime**



Beans

What is a Bean?

- A bean is an **object** that is instantiated, assembled, and otherwise managed by a Spring **IoC** container

User.java

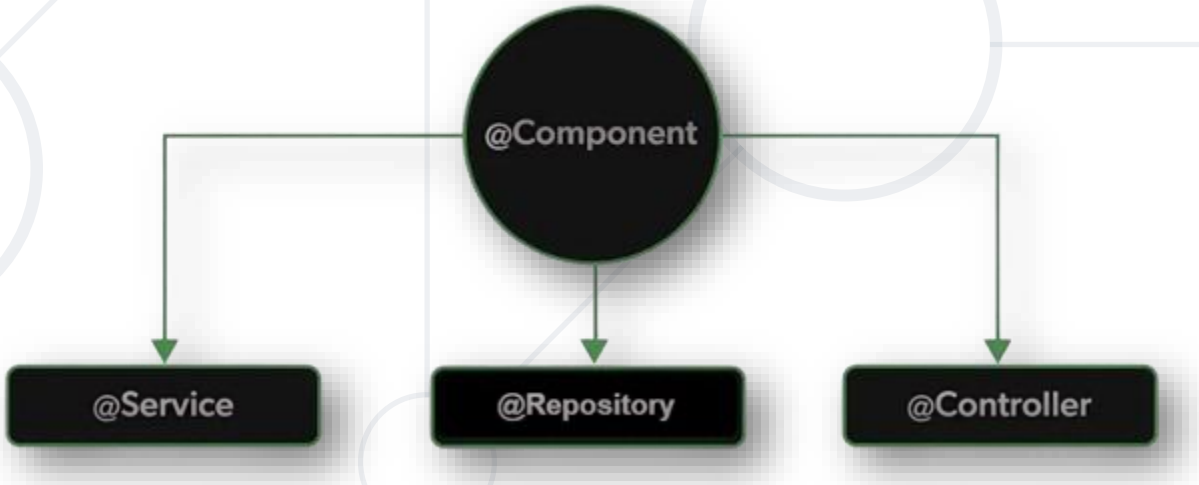
```
@Service
public class UserService {

    private final UserRepository userRepository;

    @Autowired
    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }
    // Some business methods (login, register, changeRole, etc.)
}
```

Ways of Configuring Spring Beans

- **Annotations**
  - Spring automatically detects classes annotated with **@Component**, **@Service**, **@Repository** or **@Controller**
- **Configuration Classes**
  - Use **@Configuration** classes with methods annotated by **@Bean** to define beans programmatically
- **XML Configuration**
  - Define beans and their dependencies declaratively in **XML** files

Annotations

- **@Component**: General-purpose Spring bean
- **@Service**: Indicates the class is holding the business logic
- **@Repository**: Indicates the class is dealing with database communication
- **@Controller**: Indicates the class is handling web requests



Configuration Classes

- Class annotated with **@Configuration** define Spring Beans programmatically

```java
@Configuration
public class AppConfig {

    @Bean
    public UserService userService() {
        return new UserService(userRepository());
    }

    @Bean
    public UserRepository userRepository() {
        return new UserRepository();
    }
}
```

XML Configuration

- Use **XML** files to declare beans and their dependencies in a declarative format

```xml
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
                        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="userService" class="com.example.UserService">
      <property name="userRepository" ref="userRepository"/>
    </bean>

    <bean id="userRepository" class="com.example.UserRepository"/>

</beans>
```

Get Bean from Application Context

MainApplication.java

```
@SpringBootApplication
public class Application {
  public static void main(String[] args) {
    ApplicationContext context = SpringApplication.run(Application.class, args);

    // Retrieve the UserService bean/object from the application context
    UserService userService = context.getBean(UserService.class);

    User user = userService.getById(1);
    System.out.println(user);
    }
}
```

Beans Scopes in Spring Framework

- Singleton
- Prototype
- Request
- Session
- Global Session
- Custom Scope

Singleton Scope

- Container creates a **single instance** of that bean, and all requests for that bean name will return the **same object**, which is cached
- This is **default** scope

```
@Service
@Scope("singleton") // This is redundant because Spring beans
are singleton-scoped by default

public class UserService {
    // some methods
}
```

Prototype Scope

- Will return a different **instance** every time it is requested from the container

```
@Service
@Scope("prototype") // This explicitly sets the
bean scope to prototype

public class UserService {
    // some methods
}
```

Bean Scope


Singleton
Session
Request
Spring Bean Scopes
Prototype
Global Session
Custom Scope

Application Properties

Common Application Properties

- Various properties can be specified inside your **application.yaml** file
- Property contributions can come from **additional jar files**
- You can define your **own properties**
- **Link to documentation**

Application Properties Example

application.properties

```
spring.datasource.driverClassName=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/thymeleaf_adv_lab_exam_db?c
reateDatabaseIfNotExist=true
spring.datasource.username=root
spring.datasource.password=12345
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL8Dialect
spring.jpa.properties.hibernate.format_sql = TRUE
spring.jpa.hibernate.ddl-auto = update
spring.jpa.open-in-view=false
logging.level.org = WARN
logging.level.blog = WARN
logging.level.org.hibernate.SQL = DEBUG
logging.level.org.hibernate.type.descriptor = TRACE
server.port=8000
```

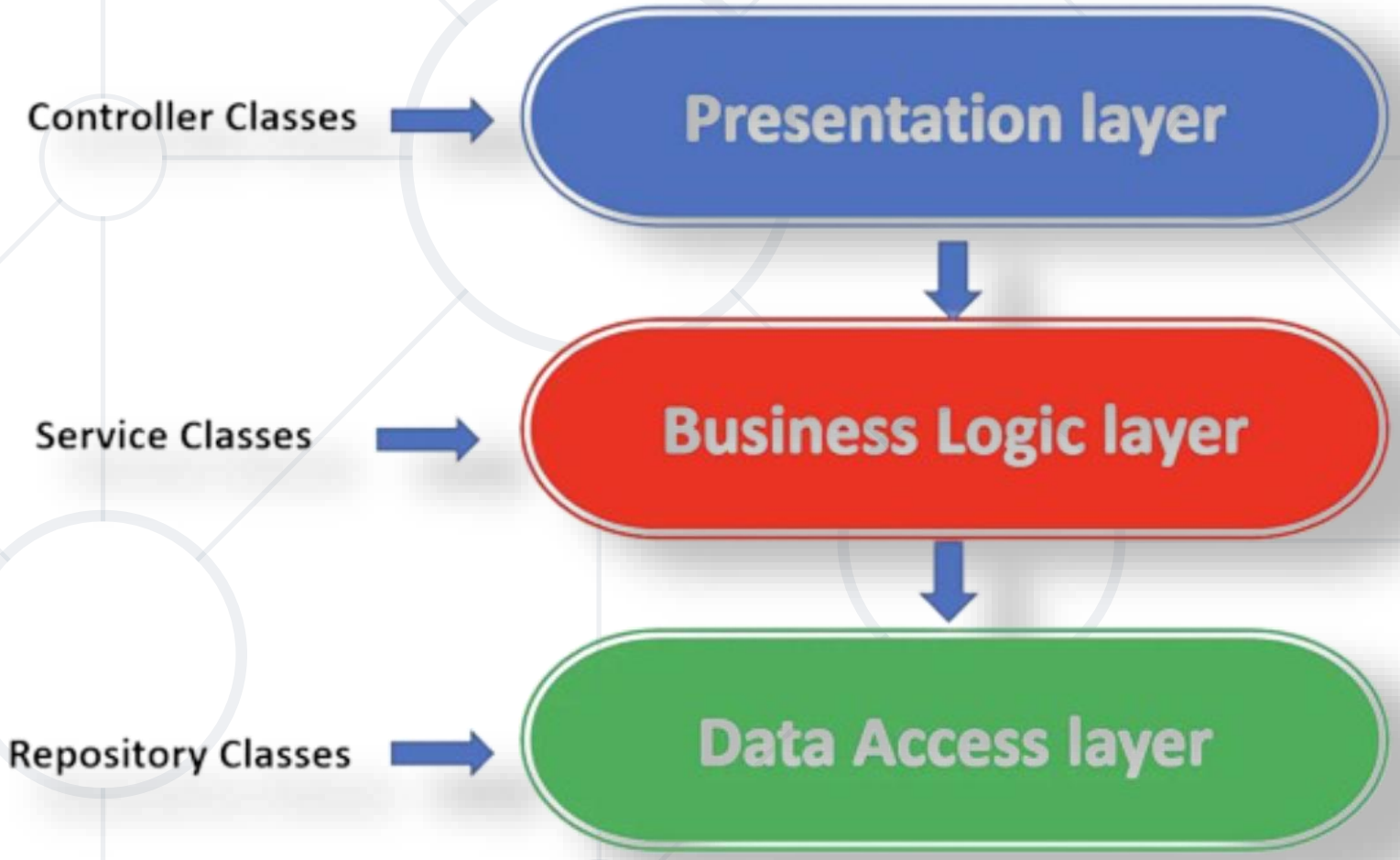
Application Yaml Example

application.yaml

```yaml
spring:
  datasource:
    driverClassName: com.mysql.cj.jdbc.Driver
    password: 12345
    url:
jdbc:mysql://localhost:3306/spring_data_lab_db?allowPublicKeyRetrieval=true&useSSL=fa
lse&createDatabaseIfNotExist=true
    username: root
  jpa:
  database-platform: org.hibernate.dialect.MySQL8Dialect
  hibernate:
      ddl-auto: create-drop
      open-in-view: false
      properties:
      hibernate:
        format_sql: true
```

Layered Architecture

Three-Tier Architecture

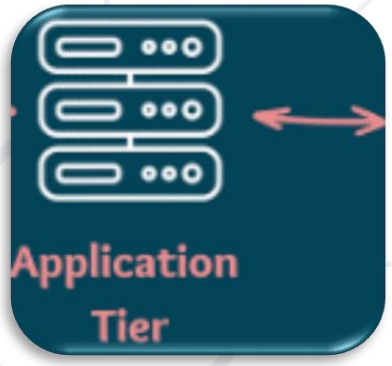


Layers in Three-Tier Architecture

- **Presentation Layer** (User Interface)
  - Handles user interactions and requests
  - Processes input and displays the output
  - Classes:
    - Controllers (**@Controller**, **@RestController**) for handling HTTP requests
    - View technologies like Thymeleaf, JSP, or REST APIs

Layers in Three-Tier Architecture

- **Business Logic Layer** (Core business logic)
  - Contains the core logic of the application
  - Acts as a bridge between the Presentation and Data Access Layers
  - Classes:
    - Service classes annotated with **@Service**
    - Contains methods that process business rules or orchestrate workflows



Layers in Three-Tier Architecture

- **Data Access Layer** (Persistence Layer)
  - Manages **communication** with the database or other external data sources
  - Handles **CRUD** (Create, Read, Update, Delete) operations
  - Classes:
    - Repository classes annotated with **@Repository**
    - Interfaces extending **JpaRepository** or **CrudRepository** in Spring Data

Layers in Three-Tier Architecture

- **Utility / Shared Layer** (Optional but Common)
  - Provides reusable utility classes shared across all layers
  - Examples:
    - **DateUtils**, **ValidationUtils** (Utility classes)
    - Exception handlers, configurations, or logging utilities

Layers in Three-Tier Architecture

```
com/example/
    ├── controller/              --> Presentation Layer
    │   ├── UserController.java
    │   ├── OrderController.java
    │
    ├── service/                 --> Service Layer
    │   ├── UserService.java
    │   ├── OrderService.java
    │
    ├── repository/              --> Data Access Layer
    │   ├── UserRepository.java
    │   ├── OrderRepository.java
    │
    ├── model/                   --> Domain Models
    │   ├── User.java
    │   ├── Order.java
    │
    ├── shared/                  --> Shared Utilities and Configurations
    │   ├── utils/               --> DateUtils.java, ValidationUtils.java
    │   ├── exception/           --> CustomException.java
    │   └── config/              --> ApplicationConfig.java
```

Feature-Based Packaging

- **Feature-Based Packaging** with Three-Tier Architecture is an **advanced variation** of the traditional Three-Tier Architecture
- **Key Difference**:
  - The **web package** is kept **outside** the feature-specific folders because a single controller can interact with **multiple services** from different features if needed
  - This approach provides flexibility and better scalability in **professional projects**

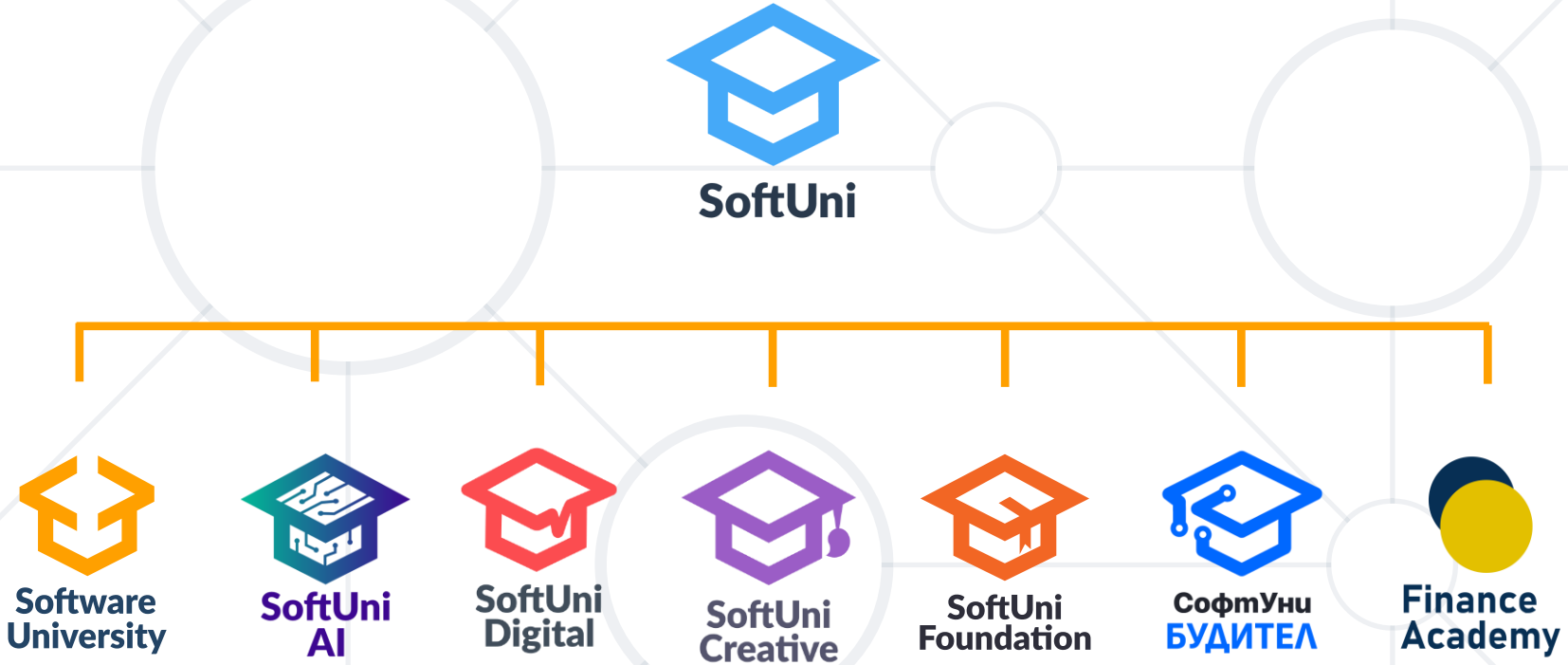
Feature-Based Packaging

```
com/example/
    ├── web/
    │   ├── controller/         --> Centralized Controllers
    │   │   ├── UserController.java
    │   │   └── OrderController.java
    │   ├── dto/                --> Data Transfer Objects (DTOs)
    │   │   ├── UserRegisterRequest.java
    │   │   ├── UserRegisterResponse.java
    │   │   ├── OrderCreateRequest.java
    │   │   └── OrderCreateResponse.java
    │   └── mapper/             --> Reusable Mappers
    │       └── DtoMapper.java
    │
    ├── user/
    │   ├── service/            --> UserService.java
    │   ├── repository/         --> UserRepository.java
    │   ├── model/              --> User.java
    │   └── property/           --> UserProperties.java
    │
    ├── order/
    │   ├── service/            --> OrderService.java
    │   ├── repository/         --> OrderRepository.java
    │   ├── model/              --> Order.java
    │   └── property/           --> OrderProperties.java
    │
    └── shared/                 --> Shared Utilities and Helpers
        ├── utils/              --> DateUtils.java, ValidationUtils.java
        ├── exception/          --> CustomException.java
        └── config/             --> ApplicationConfig.java
```

Summary

- Introduction to Spring
- What is Spring Boot?
- Inversion of Control
- Dependency Injection
- Spring Beans
- Application Properties
- Layered Architecture

Questions?



SoftUni Diamond Partners

Software University

Coca-Cola HBC Bulgaria

Postbank

top EMPLOYER
България
Bulgaria
2025
FOR A BETTER WORLD OF WORK

SUPER HOSTING .BG

encorp.ai

createX

DRAFT KINGS
THE CROWN IS YOURS

INDEAVR
Serving the high achievers

VIVACOM

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, about.softuni.bg
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity

License

- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – https://about.softuni.bg/
- © Software University – https://softuni.bg