Spring Fundamentals

SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://softuni.bg

Questions

Software University

sli.do

#java-web

Table of Contents

What is IoC Container?

Inversion of Control

- **Inversion of Control (IoC)**
  - In traditional programming, the **flow of control** and the responsibility for creating and managing objects lies with the developer
  - In the **IoC** paradigm, control over object creation and flow is **inverted**. Instead of the developer controlling the creation and lifecycle of objects, this responsibility is delegated to a framework or container

IoC Container

Software University

- **IoC Container**
  - The IoC container is a core component of **IoC-based** development frameworks like **Spring**. It manages the lifecycle of objects, their instantiation, configuration and destruction
  - The container **creates** and **manages** objects based on the configuration provided by the developer, reducing the amount of boilerplate code needed
  - The **container** acts as a factory for creating objects, a registry for managing their dependencies, and a runtime environment for executing their methods

Key Features of IoC Container

- **Key Features of IoC Container**
  - **Object Creation:** The container is responsible for creating instances of objects defined by the developer
  - **Dependency Management:** It manages the dependencies between objects, ensuring that dependencies are satisfied when objects are instantiated

Key Features of IoC Container

- **Key Features of IoC Container**
  - **Configuration:** The container allows developers to define **object** configurations either through XML configuration files, Java annotations or Java-based configuration classes
  - **Lifecycle Management:** It manages the **lifecycle** of objects, including their initialization, use and destruction
  - **Inversion of Control:** By delegating control over object creation and management to the container, IoC promotes **loose coupling** and high cohesion in the application architecture

Inversion of Control

- Spring provides **Inversion of Control** and **Dependency Injection**

UserServiceImpl.java

```
//Traditional Way
public class UserServiceImpl implements
UserService {

private UserRepository userRepository = new
UserRepositoryImpl();

}
```

UserServiceImpl.java

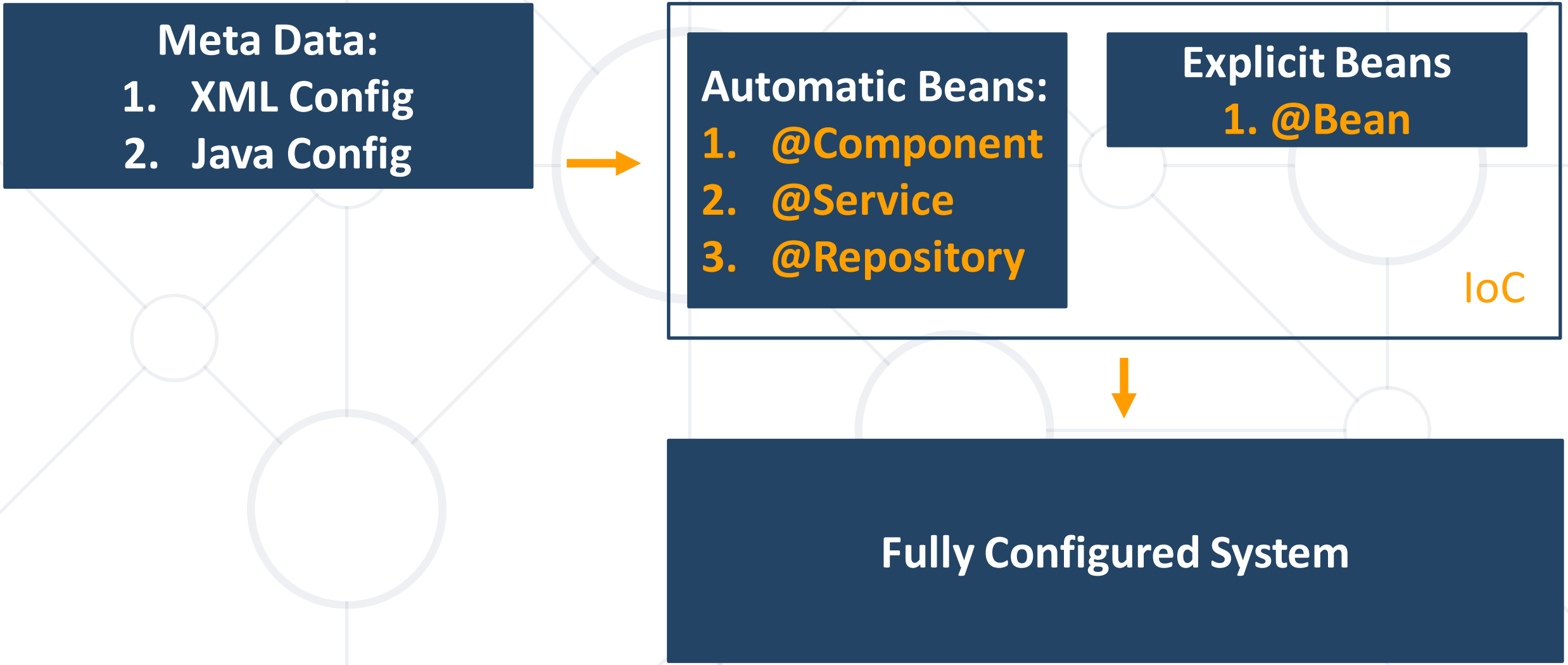
```
//Dependency Injection
@Service
public class UserServiceImpl implements
UserService {

@Autowired
private UserRepository userRepository;
}
```

Advantages of IoC Container

- **Advantages of IoC Container**
  - **Increased Modularity and Testability:** IoC promotes modularity by **decoupling** components and making them easier to test in isolation
  - **Flexibility and Extensibility:** IoC containers provide a flexible and extensible architecture that allows developers to easily plug in new components and services **without** modifying existing code
  - **Improved Maintainability:** By centralizing object management and configuration, IoC containers make it easier to **maintain** and **evolve** applications over time

Spring IoC


Meta Data:
1. XML Config
2. Java Config
Automatic Beans:
1. @Component
2. @Service
3. @Repository
Explicit Beans
1. @Bean
IoC
Fully Configured System

Beans

Beans

- Object that is **instantiated**, **assembled**, and otherwise managed by a **Spring IoC** container

Dog.java

```
public class Dog implements Animal {

    private String name;

    public Dog() {}

    // GETTERS AND SETTERS
}
```

Beans

- **Spring** manages Java objects as **beans**
- These beans are defined in configuration files or through annotations, and the container is responsible for creating instances of these beans and injecting their dependencies

MainApplication.java

```
@SpringBootApplication
public class MainApplication {
    …
    @Bean
    public Animal getDog(){
        return new Dog();
    }
}
```

Bean Declaration

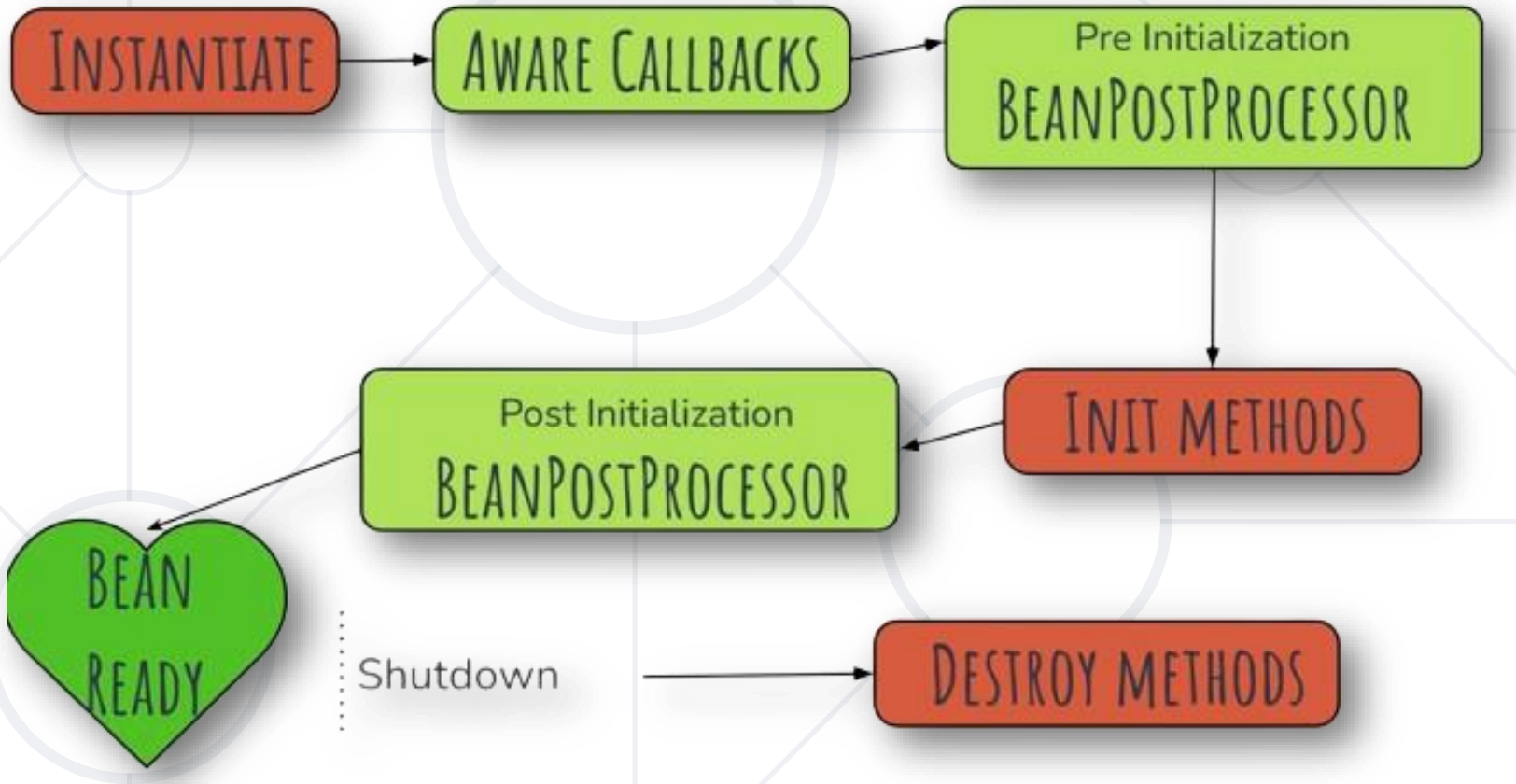
Get Bean from Application Context

MainApplication.java

```
@SpringBootApplication
public class MainApplication {
  public static void main(String[] args) {
    ApplicationContext context = SpringApplication.run(MainApplication.class, args);
    Animal dog = context.getBean(Dog.class);
    System.out.println("DOG: " + dog.getClass().getSimpleName());
    }
}
```

```
2017-03-05 12:59:19.389  INFO
2017-03-05 12:59:19.469  INFO
2017-03-05 12:59:19.473  INFO
DOG: Dog
```

Bean Lifecycle



Bean Lifecycle Demo

Software University

MainApplication.java

```
@SpringBootApplication
public class MainApplication {

 public static void main(String[] args) {
        ApplicationContext context =
       SpringApplication.run(MainApplication.class, args);
        ((AbstractApplicationContext)context).close();
 }
 @Bean(destroyMethod = "destroy", initMethod = "init")
 public Animal getDog(){
     return new Dog();
 }
}
```

Bean Lifecycle Demo

MainApplication.java

```
public class Dog implements Animal {

    public Dog() {
        System.out.println("Instantiation");
    }

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

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

```
Instantiation
Initializing..
Destroying..
```

PostConstruct Annotation

- Spring calls methods annotated with **@PostConstruct** only once, just after the initialization of bean

```
@Component
public class DbInit {
    private final UserRepository userRepository;
    public DbUnit(UserRepository userRepository)
       { this. userRepository = userRepository;}

    @PostConstruct
    private void postConstruct() {
        User admin = new User("admin", "admin password");
        User normalUser = new User("user", "user password");
        userRepository.save(admin, normalUser);
    }
}
```

PreDestroy Annotation

- A method annotated with **@PreDestroy** runs only once, just before Spring removes our bean from the application context

```
@Component
public class UserRepository {

    private DbConnection dbConnection;
    @PreDestroy
    public void preDestroy() {
        dbConnection.close();
    }
}
```

BeanNameAware Interface

- BeanNameAware makes the object aware of the bean name defined in the container

```
public class MyBeanName implements BeanNameAware {
    @Override
    public void setBeanName(String beanName) {
        System.out.println(beanName);
    }
}
```

```
@Configuration
public class Config {
    @Bean (name = "myCustomBeanName")
    public MyBeanName getMyBeanName() {
        return new MyBeanName();
    }
}
```

InitializingBean Interface

- For bean implemented **InitializingBean**, it will run **afterPropertiesSet()** after all bean properties have been set

```
@Component
public class InitializingBeanExampleBean implements InitializingBean {
    private static final Logger LOG
      = Logger.getLogger(InitializingBeanExampleBean.class);

    @Autowired
    private Environment environment;

    @Override
    public void afterPropertiesSet() throws Exception {
        LOG.info(Arrays.asList(environment.getDefaultProfiles()));
    }
}
```

DisposableBean Interface

- For bean implemented **DisposableBean**, it will run **destroy()** after Spring container is released the bean

```
@Component
public class Bean2 implements DisposableBean {

    @Override
    public void destroy() throws Exception {
        System.out.println(
          "Callback triggered - DisposableBean.");
    }
}
```

Beans Scopes in Spring Framework

- There are part of Beans scopes:
  - **Singleton**
  - **Prototype**
  - **Request**
  - **Session**

Singletone 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

```
@Bean
@Scope("singleton") <- Can be omitted
public Student student() {
    return new Student();
}
```

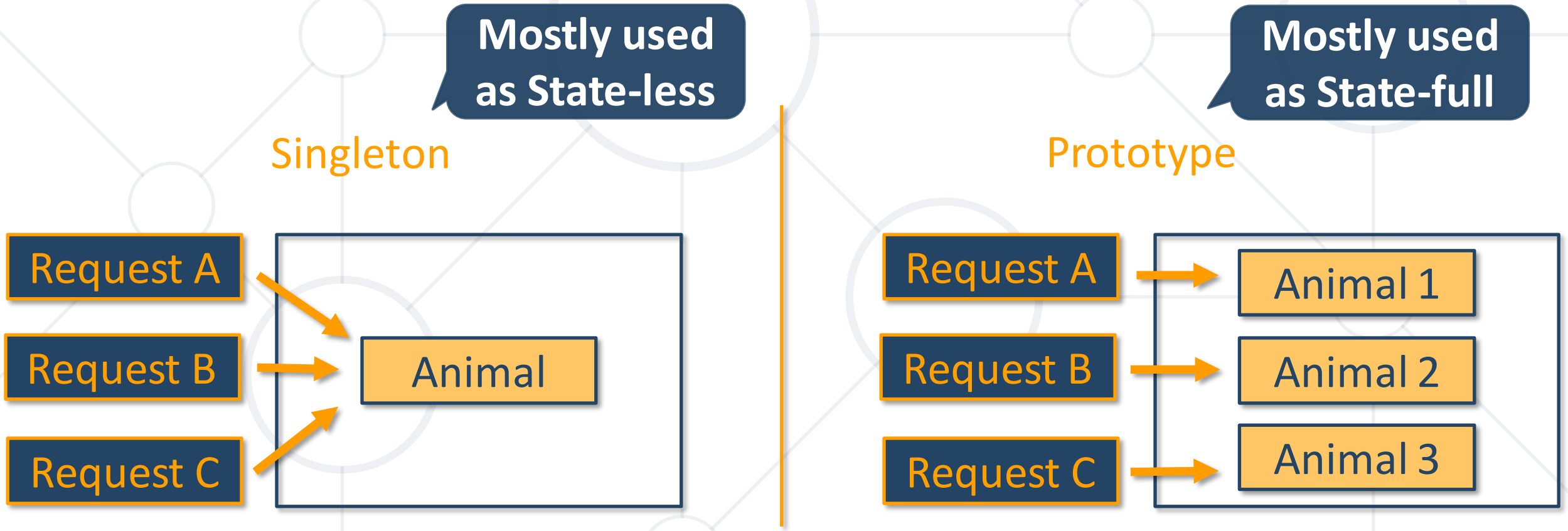
Prototype Scope

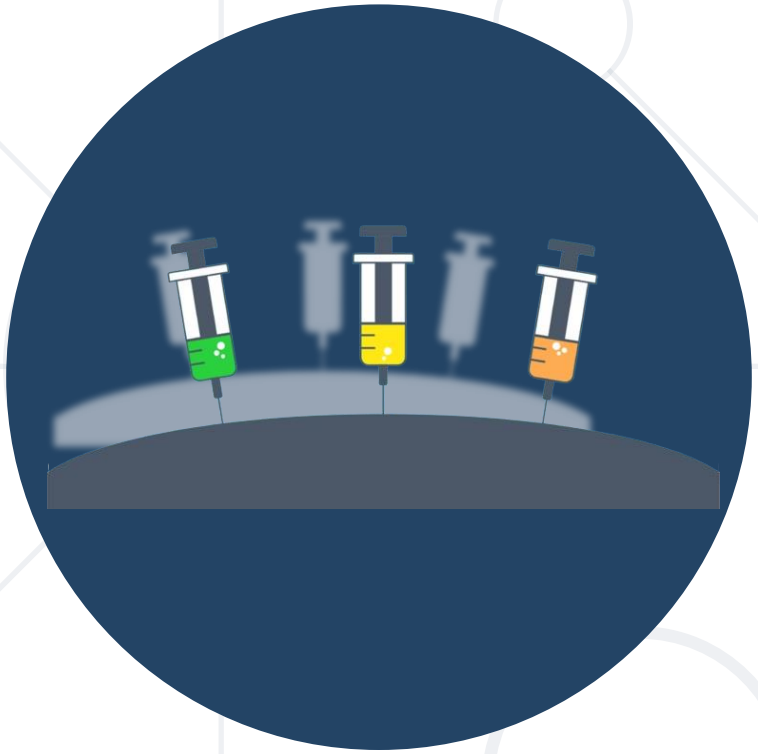
- Will return a different **instance** every time it is requested from the container

```
@Bean
@Scope("prototype")
public Student student() {
    return new Student();
}
```

Bean Scope

- The default one is **Singleton**. It is easy to change to **Prototype**





Dependency Injection

Dependency Injection

Software University

- **Dependency Injection (DI)**
  - Dependency Injection is a **design pattern** and a technique used to implement **IoC**
  - In DI, the dependencies of an object are **"injected"** into it from an external source rather than being created by the object **itself**
  - This reduces **coupling** between components, making them easier to test, reuse and maintain
  - Spring uses DI **extensively** to manage dependencies between beans, allowing you to wire them together without hardcoding dependencies in the code

Key Concepts of DI

Software University

- **Key Concepts of DI**
  - **Dependent Object:** A dependent object is a **class** or **component** that relies on other objects or services (dependencies) to perform its tasks
  - **Dependency:** A dependency is an **external** object or service that a dependent object requires to function properly. Dependencies **can be** other classes, interfaces, resources or services
  - **Dependency Injection Container:** Also known as an IoC container, it is responsible for **managing** the dependencies of objects, instantiating them and injecting their dependencies

DI types

- **Injection Types**
  - **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
  - **Constructor Injection:** Dependencies are injected via **constructor** parameters. This is the most common and recommended form of DI
  - **Setter Injection:** Dependencies are injected via setter methods on the **dependent** class

Field Injection

- Easy to write
- Easy to add new dependencies
- It **hides** potential architectural **problems**!

```
@Autowired
private ServiceA serviceA
@Autowired
private ServiceB serviceB
@Autowired
private ServiceC serviceC
```

Constructor Injection

- Harder to add dependencies
- It **shows** potential architectural problems!

```
@Autowired
public ControllerA(ServiceA serviceA, ServiceB serviceB,
ServiceC serviceC) {
  this.serviceA = serviceA;
  this.serviceB = serviceB;
  this.serviceC = serviceC;
}
```

Setter Injection

- Create setters for dependencies
- Can be combined easily with constructor injection
- Flexibility in dependency resolution or object reconfiguration!

```
@Service
public class HomeContoller(){
 //...
 @Autowired
 public void setServiceA(ServiceA serviceA) {
   this.serviceA = serviceA;
 }
}
```

Advantages of DI

Software University

- **Advantages of DI**
  - **Loose Coupling:** DI promotes loose coupling between components by **removing** direct dependencies and allowing them to be configured externally
  - **Testability:** By injecting dependencies, it becomes easier to test individual components in isolation using mock or stub objects
  - **Flexibility and Reusability:** DI allows components to be easily reused and configured for different environments **without modifying** their source code
  - **Encapsulation:** DI encourages proper encapsulation of dependencies within components, improving code organization and maintainability

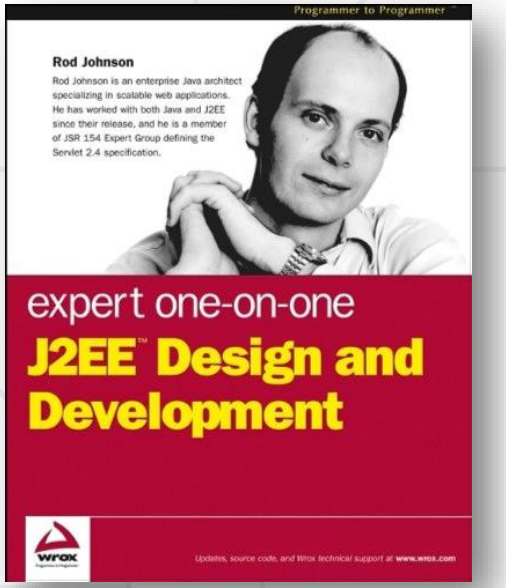
DI in Spring Framework

- **DI in Spring Framework**
  - Spring Framework provides robust support for DI through its **Inversion of Control** (IoC) container
  - **Dependencies** in Spring are typically managed using XML configuration files, Java annotations, or Java-based configuration classes
  - **Spring** supports constructor injection, setter injection, and field injection, allowing developers to choose the most suitable approach for their needs
  - The **@Autowired** annotation is commonly used in Spring to indicate dependencies that should be automatically injected by the container

What is Spring Boot?

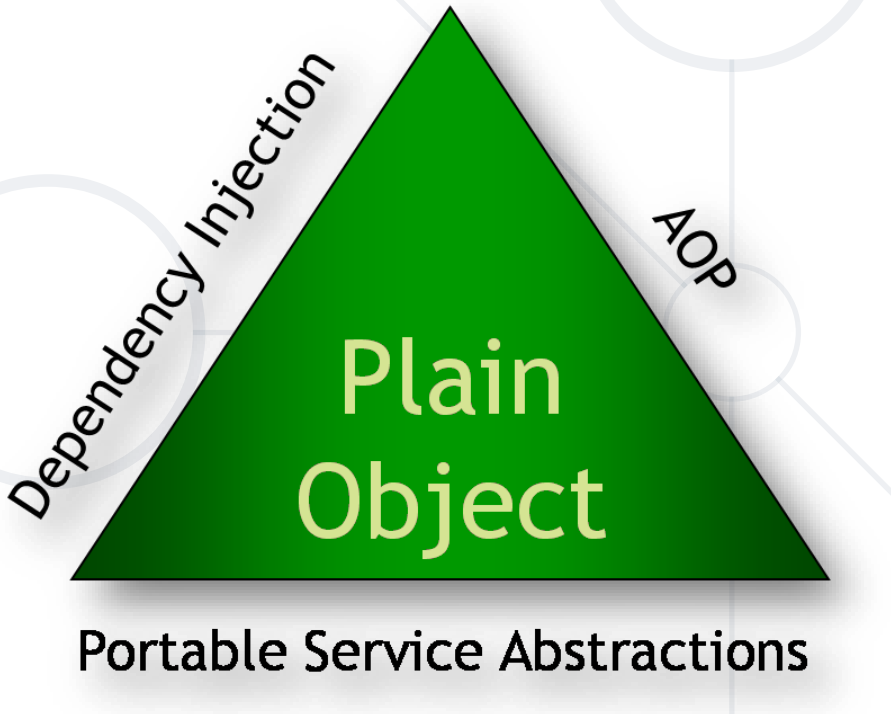
Spring

- In October **2002**, Rod Johnson wrote a book titled "Expert One-on-One J2EE Design and Development"
- The book was accompanied by **30 000** lines of framework code also known as **Interface21** (Spring 0.9)
- Since java Interfaces were the basic building blocks of **dependency injection** (DI), he named the root package of the classes as com.interface21
- Shortly after the release of the book, developers Juergen Hoeller and Yann Caroff persuaded Rod Johnson to create an open source project based on the infrastructure code. In March 2004, **Spring 1.0** was released



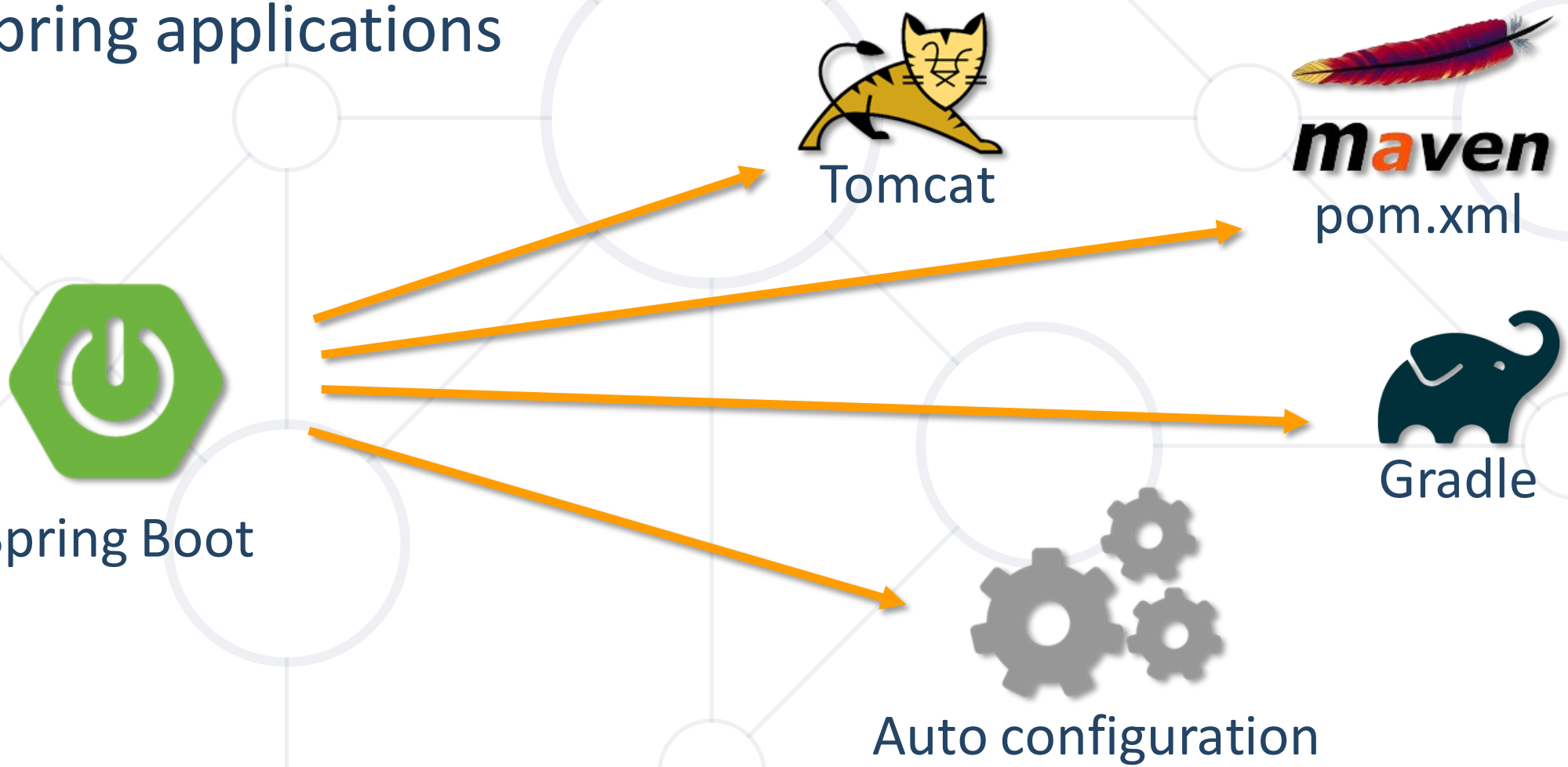
Spring Main Concepts

- **The four key concepts are:**
  - **POJO** objects
  - **Dependency Injection** (DI)
  - **AOP** (Aspect Oriented Programming)
  - Portable Service **Abstractions**



Spring Boot

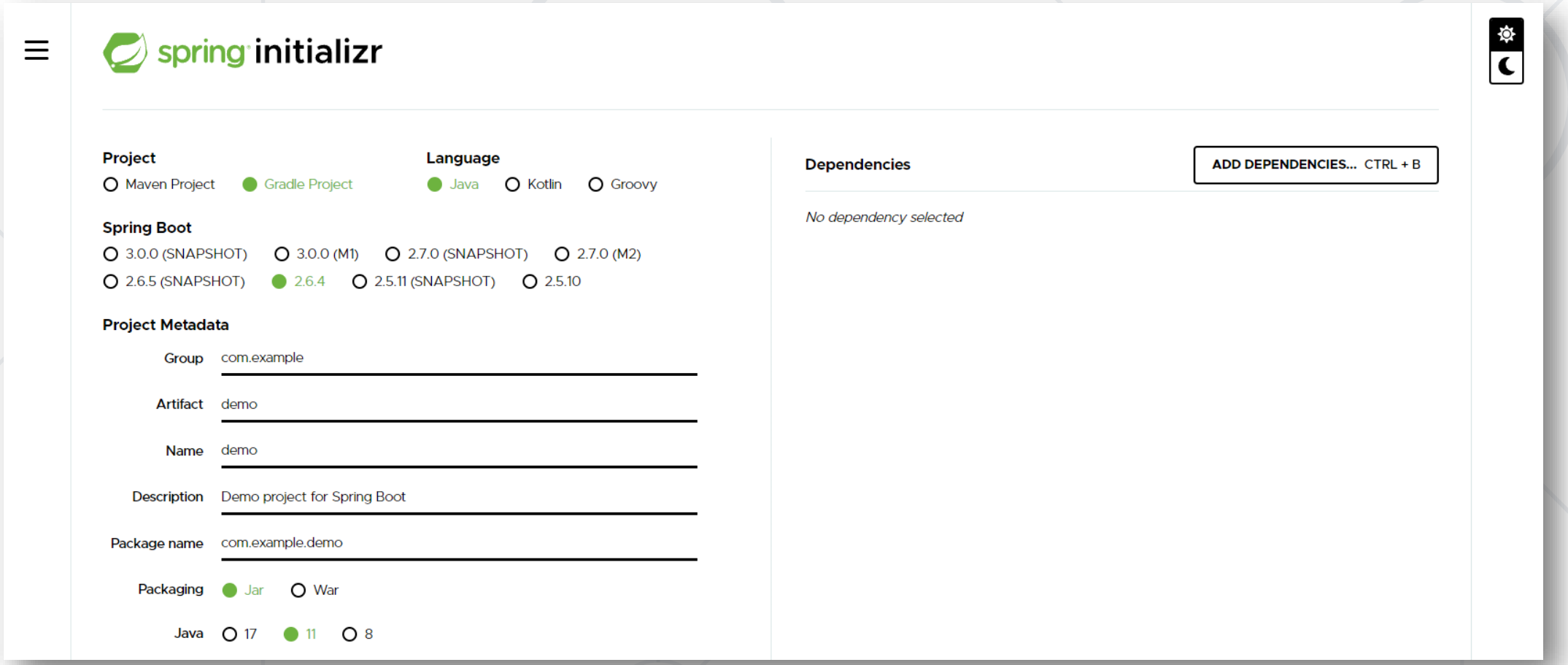
- **Opinionated view** of building production-ready Spring applications



Creating Spring Boot Project

Software University

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



Spring Dev Tools

- Additional set of **tools** that can make the application development **faster** and more **enjoyable**
- In **Maven**:

pom.xml

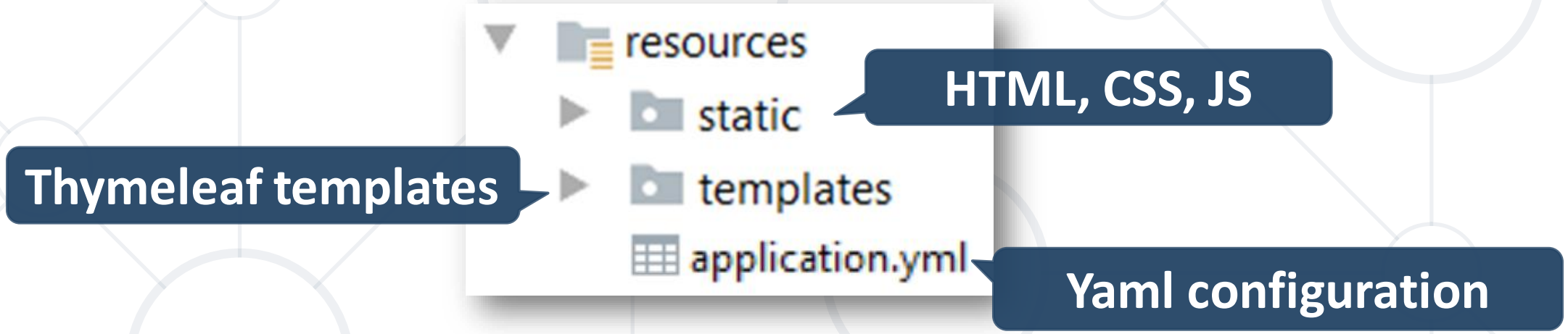
```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
</dependency>
```

- In **Gradle**:

build.gradle

```
dependencies {
    compileOnly("org.springframework.boot:spring-boot-devtools")
}
```

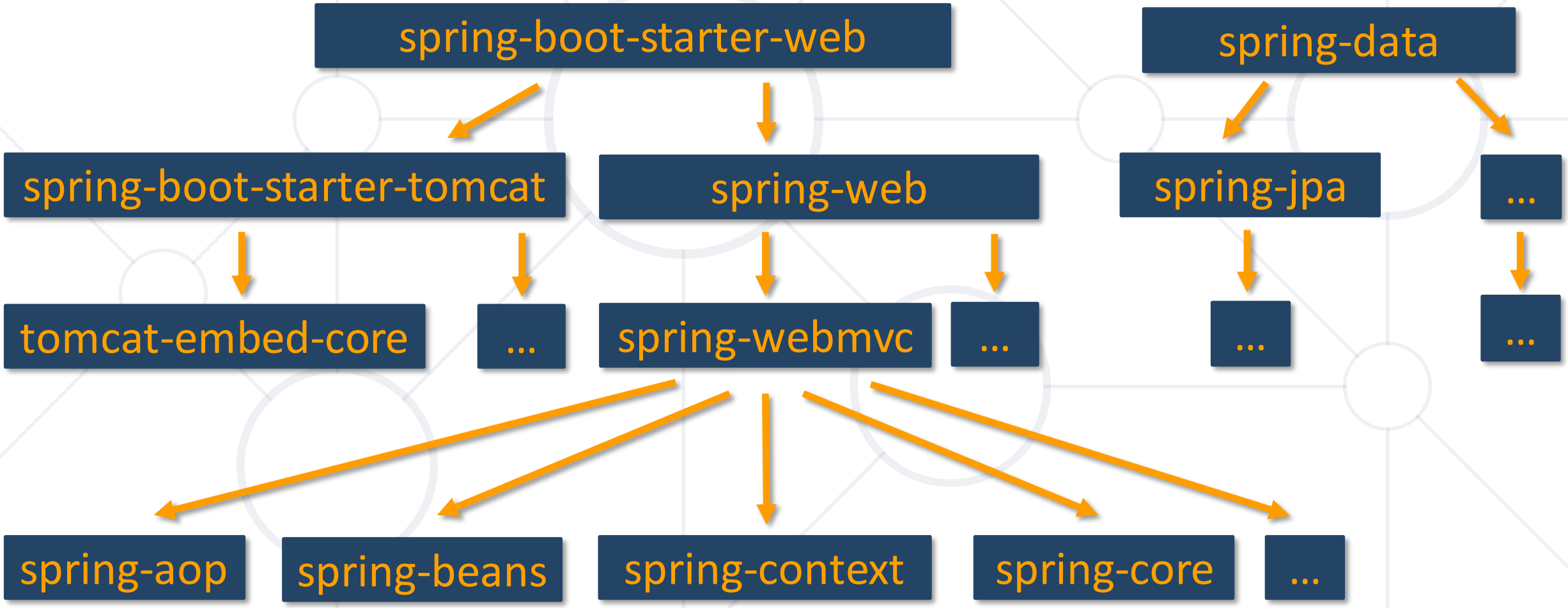
Spring Resources



Spring Boot Main Components

- Some main components:
  - **Spring Boot Starters** - combine a group of common or related dependencies into single dependency
  - **Spring Boot Auto-Configuration** - reduce the Spring Configuration
  - **Spring Boot Actuator** – provides EndPoints and Metrics
  - **Spring Data** – unify and ease the access to different kinds of database systems

Spring Boot Starters

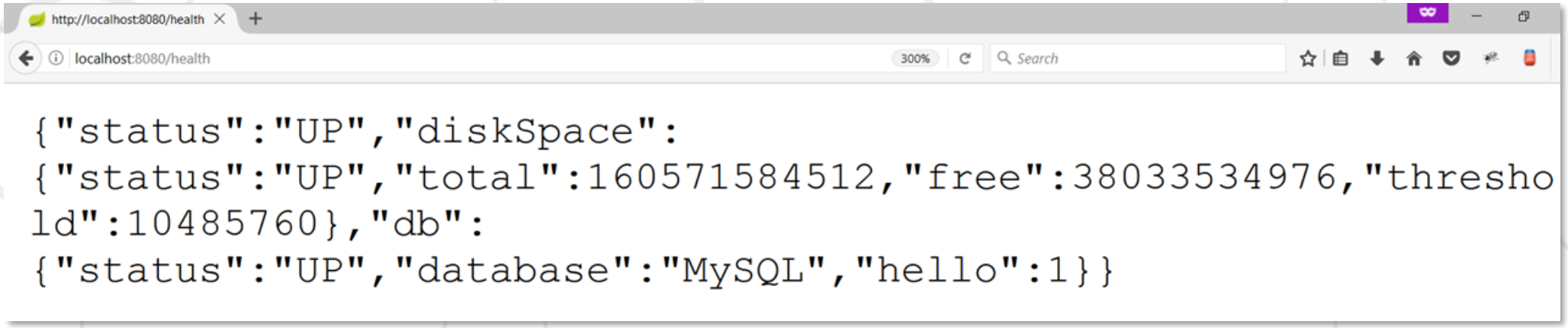


Spring Boot Actuator

- Expose different types of information about the **running application**

build.gradle

```
dependencies {
    compileOnly("org.springframework.boot:spring-boot-starter-actuator")
}
```



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

Software University

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

Software University

application.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
```

Summary

- Ioc Container
- Beans
- Dependency Injection
- Spring Boot

Questions?

SoftUni

Software University | SoftUni Creative | SoftUni Digital | SoftUni Foundation | SoftUni Kids | СофтУни БУДИТЕЛ | Finance Academy

SoftUni Diamond Partners

Software University

SmartIT

Coca-Cola HBC Bulgaria

SUPER HOSTING .BG

createX

Postbank

Решения за твоето утре

DRAFT KINGS

VIVACOM

AMBITIONED

INDEAVR

Serving the high achievers

SOFTWARE GROUP

PHAR VISION

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