cognizant

Spring Fundamentals

Emilian Marian - Software Engineer
Alexandru Babei - Software Engineer

March 2024

How Spring can improve your life as a programmer?

Agenda

1. What is Spring Framework
2. What is IoC (Inversion of Control)
3. What does Spring to make our life easier
4. How the Spring magic works
5. Configure ‘The Magic’
6. How to configure different types of Beans
7. Configure Beans scope
8. Configure Spring with external files

What is Spring?

What is Spring?

Spring is an **Inversion of Control** (IoC) container.

What is IoC example:

Own car:

- Take care of: insurance, revision, tires etc.
- If you want to change it, it will take time and an extra money effort.

Rented car:

- Rental agency will do this for you.
- If you want to change it, it will take few days, no extra money effort.

Plain code:

- You can’t decide at runtime the implementation you want to use, or it’s pretty hard to do it.
- You have to take care of everything (instantiate the class, take care of the implementation, etc.).

IoC code:

- It is easy to switch between different implementations.
- It increases the modularity of the program.

What is Spring?

IoC with Java example

```
public class Service {

  // The normal way
  private IValidator validator;
  public Service() {
    this.validator = new Validator();
  }

  // The IoC way
  private IValidator validatorIoC;
  public Service(IValidator validator) {
    this.validatorIoC = validator;
  }
```

What problem does Spring resolve and how?

First, let’s define a model.

```java
public class User {
  private String name;
  private String surname;

  public User(String name, String surname) {
    this.name = name;
    this.surname = surname;
  }
  public User() {
  }
  public String getName() { return name; }

  public void setName(String name) { this.name = name; }

  public String getSurname() { return surname; }

  public void setSurname(String surname) { this.surname = surname; }
}
```

What is a dependency?

```
public void save(User user) {
  if(validator.isValid(user)) {
    System.out.println("User " + user.toString() + " saved successfully.");
  } else {
    System.out.println("User you are trying to save contains errors.");
  }
}
```

How to instantiate dependencies with new?

```java
public class Service {
  // The normal way
  private final Validator validator;
  public Service() { this.validator = new Validator(); }

  public void save(User user) {
    if(validator.isValid(user)) {
      System.out.println("User " + user.getName() + " " + user.getSurname() + " saved successfully.");
    } else {
      System.out.println("User you are trying to save contains errors.");
    }
  }
}
```

How does everything work together?

```
public class Main {

  public static void main(String[] args) {
    Service service = new Service();
    User myUser = new User( name: "Emilian",  surname: "Marian");

    service.save(myUser);
  }
}
```

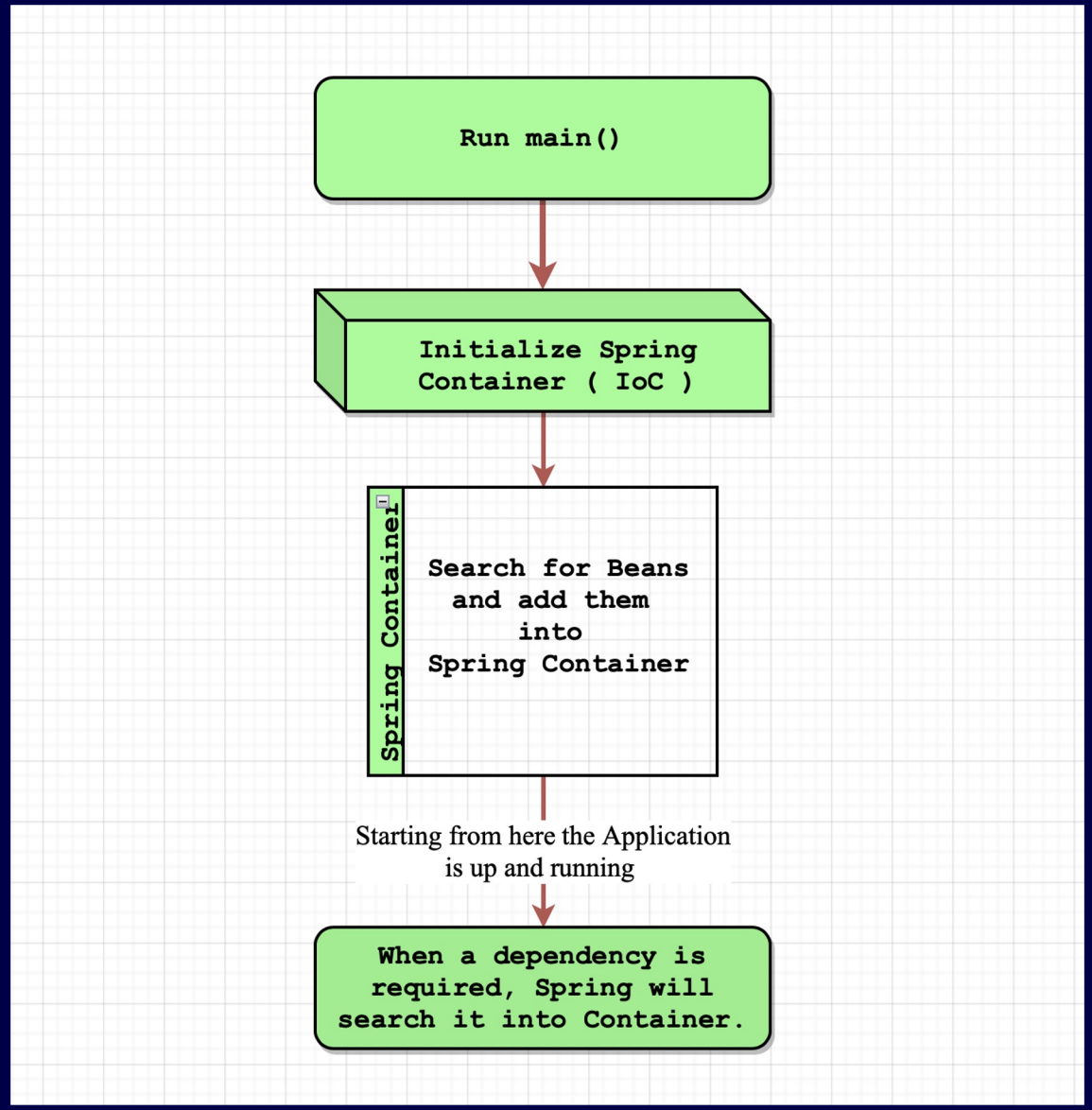
Advantages of using SPRING :

- Increase Testability.
- The code is easy to maintain.
- Easy to scale the application.
- Reduce code complexity.

How does Spring resolve this dependency problem?

- **It creates an ApplicationContext that:**
  - ApplicationContext is the implementation of IoC
  - IoC is achieved with Dependency Injection
  - Manage the Beans of an application
- **What is a Bean?**
  - A bean is an object that is instantiated, assembled and otherwise managed by a Spring IoC

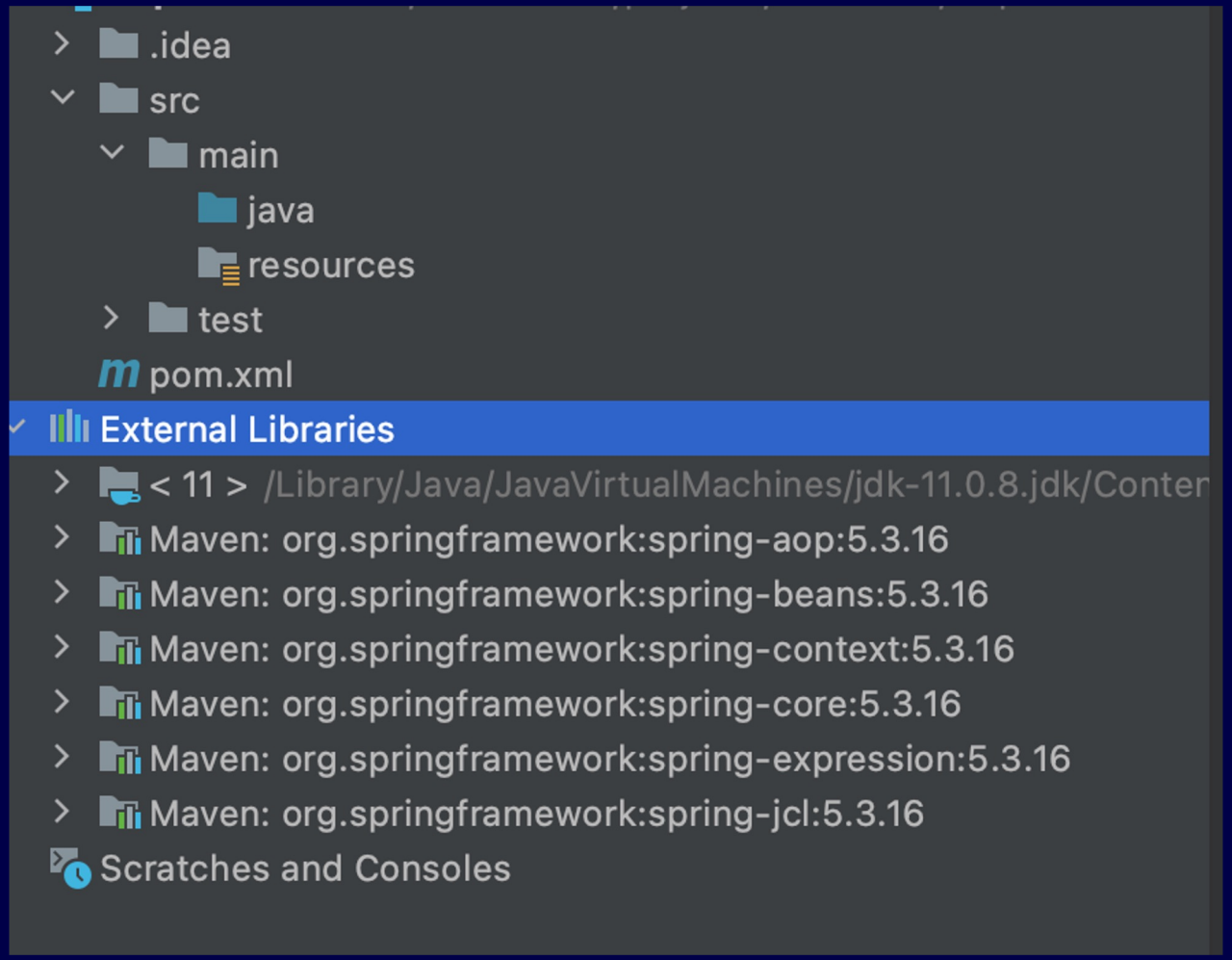
How Spring works?



Add Spring to your project.

```xml
<dependencies>
  <!-- https://mvnrepository.com/artifact/org.springframework/spring-context -->
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.3.16</version>
  </dependency>
</dependencies>
```

Add Spring to your project.



Use Spring for the code above

How an xml configuration file looks like?

Define ApplicationContext.xml under /resource file following the below structure.

```xml
<beans>
  <bean id="validatorBean" class="fii.service.Validator" />
  <bean id="serviceBean" class="fii.service.Service">
    <property name="validator" ref="validatorBean" />
  </bean>
</beans>
```

Configure application with Spring

@Configuration

- Indicates that a class declares one or more @Bean methods and may be processed by the Spring container to generate bean definitions and service requests for those beans at runtime

@Bean

- Indicates that a method produces a bean to be managed by the Spring container.
- A Bean is simply one of many objects in your application.

@Autowire

- Marks a constructor, field, setter method, or config method as to be autowired by Spring's dependency injection facilities.

Configure application with Spring

@ComponentScan

- **With this annotation we can tell Spring where to search for classes annotated with @Bean, @Service, @Component or @Repository:**

@Component - is a generic stereotype for any Spring-managed component.

@Service - annotates classes at the service layer.

@Repository - annotates classes at the persistence layer, which will act as a database repository.

Configure application with Spring

```java
@Configuration
public class ApplicationConfig {

  @Bean(name = "serviceBean")
  public Service getService() {
    return new Service(getValidator());
  }

  @Bean(name = "validatorBean")
  public IValidator getValidator() {
    return new Validator( bannedName: "John",  bannedSurname: "Doe");
  }
}
```

Configure application with Spring

```java
public class Main {

  public static void main(String[] args) {
    ApplicationContext applicationContext = new AnnotationConfigApplicationContext(ApplicationConfig.class);

    Service service = applicationContext.getBean(Service.class);
    User myUser = new User( name: "Emilian",  surname: "Marian");

    service.save(myUser);
  }
}
```

Configure application with Spring

```java
public class Service {

  @Autowired
  private IValidator validator;

  public Service() {
  }

  public void save(User user) {
    if (validator.isValid(user)) {
      System.out.println("User " + user.getName()
          + " " + user.getSurname() + " saved successfully.");
    } else {
      System.out.println("User you are trying to save contains errors.");
    }
  }
}
```

```java
import org.springframework.context.annotation.ComponentS
import org.springframework.context.annotation.Configurat

@Configuration
@ComponentScan({"fii.service"})
public class ApplicationConfig {

  @Bean(name = "serviceBean")
  public Service getService() { return new Service(); }

  @Bean(name = "validatorBean")
  public IValidator getValidator() {
    return new Validator( bannedName: "John",
        bannedSurname: "Doe");
  }
}
```

Let’s make some examples together 

Spring bean scopes

Bean scopes

- The latest version of the Spring framework defines 6 types of scopes:
  - singleton
  - prototype
  - request
  - session
  - application
  - websocket

- Request, session, application and websocket scopes are only available in a web-aware application.

Bean scopes

Singleton

```
@Bean
@Scope(BeanDefinition.SCOPE_SINGLETON)
public UserService getUserService() {
    return new UserService();
}

@Bean
@Scope("singleton")
public UserService getUserServiceWithStringScope() {
    return new UserService();
}
```

Prototype

```
@Bean
@Scope(BeanDefinition.SCOPE_PROTOTYPE)
public UserService getUserService() {
    return new UserService();
}

@Bean
@Scope("prototype")
public UserService getUserServiceWithStringScope() {
    return new UserService();
}
```

Spring configurations

Environment properties

```java
public class MyApplication {

    public static void main(String[] args){
        ApplicationContext ctx = new AnnotationConfigApplicationContext(someConfigClass);
        Environment env = ctx.getEnvironment();
        String databaseURL = env.getProperty("database.url");
        boolean containsPassword = env.containsProperty("database.password");
    }
}
```

Resources

```
public class MyApplication {

    public static void main(String[] args){
        ApplicationContext ctx = new AnnotationConfigApplicationContext(someConfigClass);

        Resource aClasspathTemplate = ctx.getResource("classpath:somePackage/application.properties");
        Resource aFileTemplate = ctx.getResource("file://someDirectory/application.properties");
        Resource anHttpTemplate = ctx.getResource("https://cognizant.com/jobs");
        Resource s3Resource = ctx.getResourc("s3://myBucket/myFile.txt");
    }
}
```

Property sources

- Spring 3.1 introduced the **@PropertySource** annotation as a convenient mechanism for adding property sources to the environment.
- Another very useful way to register a new properties file is using a **placeholder**, which allows us to dynamically select the **right file at runtime**
- The @PropertySource annotation is repeatable according to Java 8 conventions. Or you can use the **@PropertySources** annotation and specify an array of @PropertySource.

```
6   @Configuration
7   @ComponentScan({"com.myproject"})
8   @PropertySource("application.properties")
9   @PropertySource("${envTarget:dev}-application.properties")
10  public class PropertiesConfig {
11  }
```

Using/Injecting Properties

- Injecting a property with the **@Value** annotation is straightforward

- You can also specify a **default** value for the property

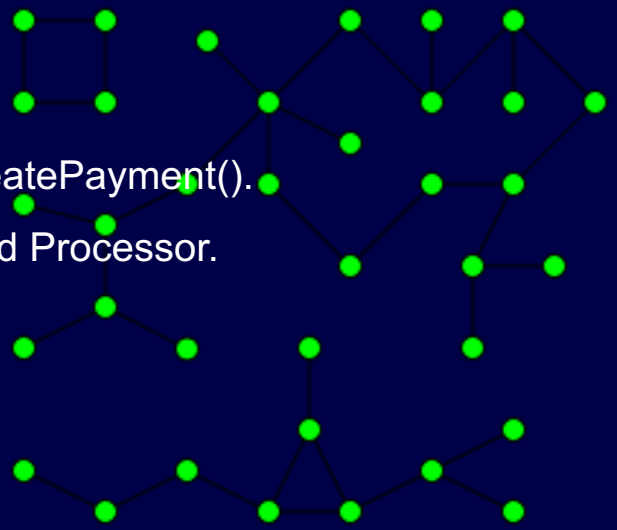
```
public class PropertiesConfig {

    @Value( "${app.property1}" )
    private String firstAppProperty;

    @Value( "${app.property2}" )
    private int secondAppProperty;

    @Value( "${app.property3:default}" )
    private String thirdAppProperty;
}
```

Let’s make some examples together 

Now, we know you can do it!

Exercise

Below you will find some tasks in a random order. Once everything it's tight together, Spring will do the magic.



1. Create a simple Java class PaymentCreationServiceImpl representing a service.
2. Define an interface PaymentCreationServiceInterface for the service with one method called createPayment().
3. Use Spring to inject PaymentService and PaymentHistory dependencies into a new class named Processor.
4. Configure beans with Annotations into PaymentConfiguration
5. In the processor, inject one dependency in the constructor and another one with @Autowire.
6. Create one main class where you can test createPayment().
7. Create a PaymentHistory class with a method in it.
8. Inject a property called `app.history.type` in PaymentHistory from application.properties and print it in the method you created.
9. Annotate all the classes with the required annotation and make your application scan for all the beans.

10. Now that everything works, you can test @Component, @Service, @Repository or you can change the @Scope, or also, you can inject your dependency in the setter.

Thank you

Emilian Marian

https://www.linkedin.com/in/emilian-marian-48535a6b

emilian.marian@cognizant.com

Alexandru Babei

alexandru.babei@cognizant.com