

Spring

**INTERA**

# Agenda

- What is Spring?
- IoC and DI (A run thru some small example applications)
- Spring & Context
- Spring Boot
- Spring Boot Configuration
- Spring Boot MVC

# What is Spring?

The Spring Framework is an application framework and inversion of control container for the Java platform

— *Wikipedia* <sup>1</sup>

---

<sup>1</sup> [https://en.wikipedia.org/wiki/Spring\\_Framework](https://en.wikipedia.org/wiki/Spring_Framework)

# Yes - but what is Spring?

Core spring is based on the ideas of Inversion of Control (IoC) and Dependency Injection (DI) - so we'll start there.

# Dependency Injection (DI)

A class is provided with the services etc that it needs rather than creating them.

## Inversion of Control (IoC) - in Spring

IoC is a very open design principle - but in Spring terms it mostly refers to the spring container that provide the actual DI mechanics (creation of beans, injecting them following configuration etc).

# IoC and DI applications

# Initial code

We start with a simple application <sup>2</sup>

---

<sup>2</sup>./exercises/exercise1/

# Services

- Calculator
- Display



## Calculator Service

```
public int plus(int a, int b);
```

```
public int minus(int a, int b);
```

```
public int multiply(int a, int b);
```

```
public int divide(int a, int b);
```

## Display Service

```
public void output(String value);
```

# Business Logic

The calculation class performs a business operation using the services.

However - let's take a look at the code:

## Main method in the Business Logic

```
public void complexCalculation() {  
    // Service 1  
    Calculator calculator = new Calculator();  
  
    int result = calculator.plus(2, 3);  
  
    // Service 2  
    Display display = new Display();  
  
    display.output(String.format("2 + 3 = %d", result));  
}
```

# Problems

- How do we test different implementations of either service?
- How do we even provide different implementations?

All of these require editing the business logic class.

# Dependency Injection

Let's take a look at how we can manually change this over to a DI based setup.

First round - manual DI - no spring.

# Constructor vs Setter

We can do this in two ways:

Provide (inject) the required services (dependencies) via:

- the constructor
- setters

# Setter injection

```
private Calculator calculator;  
private Display display;  
  
public void setCalculator(Calculator calculator) {  
    this.calculator = calculator;  
}  
  
public void setDisplay(Display display) {  
    this.display = display;  
}
```



# Constructor injection

```
private final Calculator calculator;  
private final Display display;  
  
public Calculation(Calculator calculator, Display display) {  
    this.calculator = calculator;  
    this.display = display;  
}
```

# Orchestration

OK - but how do we set up (or orchestrate) the application?

# Orchestration - Setter injection

```
public static void main(String[] args) {  
    // Services  
    Calculator calculator = new Calculator();  
    Display display = new Display();  
  
    // Setter injection  
    Calculation calculation = new Calculation();  
    calculation.setCalculator(calculator);  
    calculation.setDisplay(display);  
  
    // Business logic  
    calculation.complexCalculation();  
}
```

# Orchestration - Constructor injection

```
public static void main(String[] args) {  
    // Services  
    Calculator calculator = new Calculator();  
    Display display = new Display();  
  
    // Constructor injection  
    Calculation calculation = new Calculation(calculator, display);  
  
    // Business logic  
    calculation.complexCalculation();  
}
```

# Exercise 1

- Convert the simple initial application to be constructor injected.

# Exercise 1 - Walkthrough

First round - manual DI - no spring <sup>3</sup>

---

<sup>3</sup>./exercises/exercise1/

# Spring?

So far we have seen DI but had to orchestrate the application by hand.

Spring provides an IoC container - objects define what they need and the IoC container can then provide the required dependencies via DI.

We'll look at three ways:

- Old style (spring - with XML configured beans)
- Annotation style (spring - with annotated classes)
- Spring Boot

# Spring Beans

A spring bean is any object that is managed by the Spring IoC container.

A spring bean is usually a singleton (this is the default bean scope - we will look at scopes later on).



# Spring - XML

First steps are to grab some java libraries. We state our dependencies in the pom.xml file used by maven:

```
<dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <version>5.3.16</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.3.16</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-beans</artifactId>
    <version>5.3.16</version>
  </dependency>
</dependencies>
```

# Application context

Spring provides a set of classes (based around BeanFactory) that allows us to configure the IoC container.

However - in nearly every project it is far far more common to use spring's application context for this.

# applicationContext.xml - setter injection

```
<?xml version="1.0" encoding="UTF-8"?>
<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="display" class="com.itera.spring.Display"/>
  <bean id="calculator" class="com.itera.spring.Calculator"/>

  <bean id="calculation" class="com.itera.spring.Calculation">
    <property name="calculator" ref="calculator"/>
    <property name="display" ref="display"/>
  </bean>
</beans>
```

# applicationContext.xml - constructor injection

```
<?xml version="1.0" encoding="UTF-8"?>
<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="display" class="com.itera.spring.Display"/>
  <bean id="calculator" class="com.itera.spring.Calculator"/>

  <bean id="calculation" class="com.itera.spring.Calculation">
    <constructor-arg name="calculator" ref="calculator"/>
    <constructor-arg name="display" ref="display"/>
  </bean>
</beans>
```

# Using the context

```
// Load the context
ApplicationContext context =
    new ClassPathXmlApplicationContext("applicationContext.xml");

// Get a bean by type
Calculation calculation = context.getBean(Calculation.class);

// Get a bean by name
Calculation calculation =
    (Calculation) context.getBean("calculation");
```

## Exercise 2

- Complete the spring XML configuration for the application

Things to note - the Service classes are identical to those used in the previous exercise.

The only changes here are in how we orchestrate the app.

# Exercise 2 - Walkthrough

We'll be using spring's context and beans <sup>4</sup>

---

<sup>4</sup> [./exercises/exercise2/](#)

# Problems

This works - but - it means that the XML file is tightly coupled to the class structures.

If we change the java code we have to remember to adjust this file.



# Spring - Annotations

Let's modify the previous version using spring's component scanning mechanism.

Scanning is enabled in the application context file.

It triggers spring to go through all classes in a given package tree looking for annotations.

# Application context

The context file becomes a lot smaller - it simply configures what packages to scan

# applicationContext.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">

    <context:annotation-config/>
    <context:component-scan base-package="com.itera.spring"/>
</beans>
```

# Annotating classes

Classes get a class level annotation stating what sort of bean they are (@Service, @Component, @Repository)

Injection points are often marked @Autowired<sup>5</sup>

```
@Component  
class ServiceName {  
    ...  
}
```

---

<sup>5</sup> From Spring 4.3 a spring bean class with only one constructor does not need the autowired annotation - spring will wire it

# Using the context

The code in *Application* is exactly the same as for the *XML* version

## Exercise 3

- Annotate the two service classes and the calculation class so that the application functions.
- Consider what annotation to use in each case.

# Exercise 3 - Walkthrough

Let's modify the previous version using spring's component scanning mechanism (annotations) <sup>6</sup>

---

<sup>6</sup> [./exercices/exercise3/](#)

# What is the difference between the annotations

`@Component` - the basic spring bean marker. This is what component scanner is looking for

`@Service` - a special case of `Component` - used to state that this is a bean used in the service layer - there is no functional difference to `Component`

`@Repository` - also a special case of `Component` - but it has an extra job - to catch any persistence specific exception and to re-throw it as a standard spring exception. Requires an instance of `PersistenceExceptionTranslationPostProcessor` bean in the context <sup>6.1</sup>.

---

<sup>6.1</sup> Spring Boot adds this for you automatically



# Notes

These examples are very simple. some other things we need to consider are

- bean scopes (is it a singleton? etc)
- qualifiers (requiring a bean and there are multiple implementations available)

# Problems

- Still a lot of boiler plate
- Managing dependencies in a larger project is still challenging

# Spring Boot

Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run".  
We take an opinionated view of the Spring platform and third-party libraries so you can get started with minimum fuss. Most Spring Boot applications need minimal Spring configuration.

— *Spring.io* <sup>7</sup>

---

<sup>7</sup> <https://spring.io/projects/spring-boot/>

Spring Boot tries to simplify:

- Setup
- Dependency Management
- Configuration

# Spring Boot Starters

Spring Boot provides different starters - so that you can add support for different functionality.

We'll take a look at what's available after we've looked at the same test app in a Spring Boot version.

# Exercise 4

See README in the exercise directory.

## Exercise 4 - Walkthrough

- Classes keep the same annotations as before
- Main class gets annotated `@SpringBootApplication`
- Implement the `CommandLineRunner` as it is a command line app<sup>8</sup>

---

<sup>8</sup> `./exercises/exercise4/`

```
@SpringBootApplication
public class Application implements CommandLineRunner {
    private final ApplicationContext context;

    public Application(ApplicationContext context) {
        this.context = context;
    }

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

    @Override
    public void run(String... args) {
        Calculation calculation = context.getBean(Calculation.class);

        calculation.complexCalculation();
    }
}
```

# Common context issues



Spring complains if it cannot build a valid context

Usually it will be one of two issues:

- Cannot find a bean it needs
- Finds more than one match

# How to fix

First - dig down through the stack trace - spring will try and tell you what it didn't manage to do.

Things to remember:

- Missing annotation on a @Component or @Service or similar?
- Missing configuration or auto configuration?
- Search by type (interface) or name can give more than one hit - can you use @Qualifier?
- Component scanning also scans dependencies (if the package name is correct)
  - did you get more than you bargained for?
  - did something that was included expect certain dependencies that are not available?

# More on Spring Beans

Spring beans have a scope which defines lifecycle

- singleton (default)
- prototype

Spring web-aware only

- request
- session
- application
- websocket

## Singleton bean

The standard spring bean.

The spring container will always return the same bean.

## Prototype bean

The spring container will return a new instance every time.

# Web aware

## Lifetime of web aware beans

- request - single http request
- session - http session
- websocket - a websocket
- application - servlet context

# Spring Boot Configuration

Spring boot has a flexible approach to loading configuration properties.

For this course we will use the default application.properties file that spring initializr generates for us and simple use of @Value

You may wish to read up on:

- properties
- yaml
- @Configuration and @ConfigurationProperties
- @PropertySource
- Profiles

# Using property values

Simplest with @Value injection

```
@Value( "${config.property.name}" )  
private String configProperty;
```

## Example

Let's add a property to exercise 3:

`application.properties`:

`calculation.heading=Calculation Result:`



## Inject into Calculation and send to display:

```
public Calculation(Calculator calculator,  
                  Display display,  
                  @Value("${calculation.heading}") String heading) {  
    this.calculator = calculator;  
    this.display = display;  
    this.heading = heading;  
}  
  
public void complexCalculation() {  
    int result = calculator.plus(2, 3);  
  
    this.display.output(this.heading);  
    this.display.output(String.format("2 + 3 = %d", result));  
}
```

# Spring Boot MVC

- Resources
- Requests/sessions
- Responses

Add the web starter:

```
<dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-web</artifactId>  
</dependency>
```

# Resources

## Get all items

```
@RestController
public class ExampleController {
    private final SomeService service;

    public ExampleController(SomeService service) {
        this.service = service;
    }

    @GetMapping("/")
    @ResponseBody
    public List<Example> getAllExamples() {
        return service.examples();
    }
}
```

# PathVariable

GET /3

```
@GetMapping("/{id}")  
@ResponseBody  
public Example getExample(@PathVariable Integer id) {  
    return service.example(id);  
}
```

# RequestParam

GET /?id=3

```
@GetMapping("/")  
@ResponseBody  
public Example getExample(@RequestParam Integer id) {  
    return service.example(id);  
}
```

RequestParam can also retrieve from form posts and file uploads

# RequestBody

```
@PostMapping("/")  
@ResponseBody  
public Example addExample(@RequestBody Example example) {  
    return service.addExample(example);  
}
```

# Exercise 5

See README in the exercise directory.

# Exercise 5 - Walkthrough

Let's take a look at an example project.<sup>9</sup>

This time in kotlin with gradle using the kotlin DSL - just for fun.

Initially created with spring initializer by choosing kotlin and gradle on <https://start.spring.io/>

---

<sup>9</sup>./exercises/exercise5/



# Further Reading

- Spring presentation (full) - <https://github.com/itera/spring> - mostly the same as this with a section on reactive java/spring and a section on databases
- Test presentation - <https://github.com/Itera/java-test>
- Spring Auto-configuration
- Spring Security / OAuth
- Rest Repositories
- Spring Web Services (XML/SOAP)
- Spring Cloud
- [Project Reactor](#) (reactive java - Mono/Flux)

Many other useful sites out there - my current goto is

<https://www.baeldung.com/>