Introduction to Spring

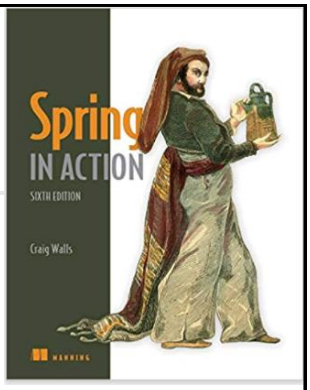
1

1

2

2

Book



Spring in Action, Sixth Edition

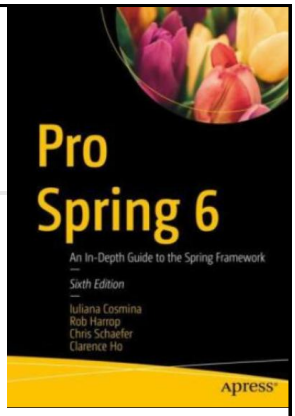
Craig Walls

A new edition of the classic bestseller! *Spring in Action, 6th Edition* covers all of the new features of Spring 5.3 and Spring Boot 2.4 along with examples of reactive programming, Spring Security for REST Services, and bringing reactivity to your databases. You'll also find the latest Spring best practices, including Spring Boot for application setup and configuration.

3

3

Book



Pro Spring 6: An In-Depth Guide to the Spring Framework

Authors: Iuliana Cosmina , Rob Harrop , Chris Schaefer , Clarence Ho

A "tried and true" in-depth guide on the Spring Framework 6 by the leading publisher of Spring books

Build complex, enterprise level cloud-native applications with Spring

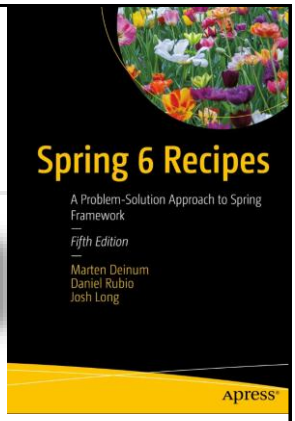
Written by Spring experts and insiders

4

4

Book

Spring 6 Recipes
A Problem-Solution Approach to Spring Framework



Authors: Marten Deinum , Daniel Rubio , Josh Long

An in-depth developer code reference on the Spring Framework 6 and related projects

Provides hundreds of reusable code snippets that can be used as templates for your own Spring applications

Includes Spring Native, R2DBC, more WebFlux and other reactive Spring application development

5

5

1. Spring Framework

WIKIPEDIA
The Free Encyclopedia

The **Spring Framework** is
an **open source application framework** and
Inversion of Control container
for the **Java** platform.

Spring

6

1. Spring Framework

First version:

Rod Johnson (October **2002**)

First released:

(June **2003**)

→ Spring 2 → … → **Spring 6.2**

1.1 Why Use the Spring Framework?

Frameworks, such as Struts, EJB2, Hibernate, …

→ forced developers to write classes
littered with **unnecessary code**,
locked into their framework
difficult to write **tests**

→ Example (EJB2):

```
public class HelloWorldBean implements SessionBean {
        public void ejbActivate(){}
        public void ejbPassivate(){}
        public void ejbRemove(){}
        public void ejbCreate(){}
        public void setSessionContext(SessionContext ctx){}
        public String sayHello(){return "Hello World";}
}
```

1.1 Why Use the Spring Framework?

→ Example (Spring):

```
public class HelloWorldBean {
        public String sayHello(){
                return "Hello World";}
}
```

POJO (Plain Old Java Objects)

- simple form
- easy to test
- powerful
  - by assembling them using **dependency injection**

9

9

Spring

1.1 Why Use the Spring Framework?

Simplifying Java Development

Spring framework is developed to **simplify** the developed of *enterprise applications* in *Java* technologies.

10

10

1.1 Why Use the Spring Framework?

Spring does **not reinvent the wheel**.

Despite its **broad scope**, Spring does not introduce its own solution (Hibernate, JPA, Web services, Ajax, JSF, JUnit, Agile Development, …).

Spring does make these existing solutions significantly easier to use, and places them in a consistent architectural approach.

11

11

1.1 Why Use the Spring Framework?

You can build **any application** in Java
stand-alone,
Web, …

12

12

2. The core of the Spring Framework

The **core** of the Spring Framework

Inversion of control
= Hollywood Principle (Don't call me, I'll call you)
Dependency Injection
JavaBeans and interfaces

Inversion of Control refers to the generally desirable architectural pattern of having an outside entity (the container) **wire** together objects, such that objects are given their **dependencies** by the container, instead of directly instantiating them themselves.

13

13

2.1 Dependency Injection

Dependency Injection:

-***dependency***: class A need class B to get its job done → class A is **dependent** on class B.

- ***injection***: class B will get **injected** into class A by the *IoC container*.

Two injection styles:

-***via arguments*** passed to the constructor when an object is created (**constructor injection**).

- ***via the setter method*** , after the object has been created (**setter injection**).

14

14

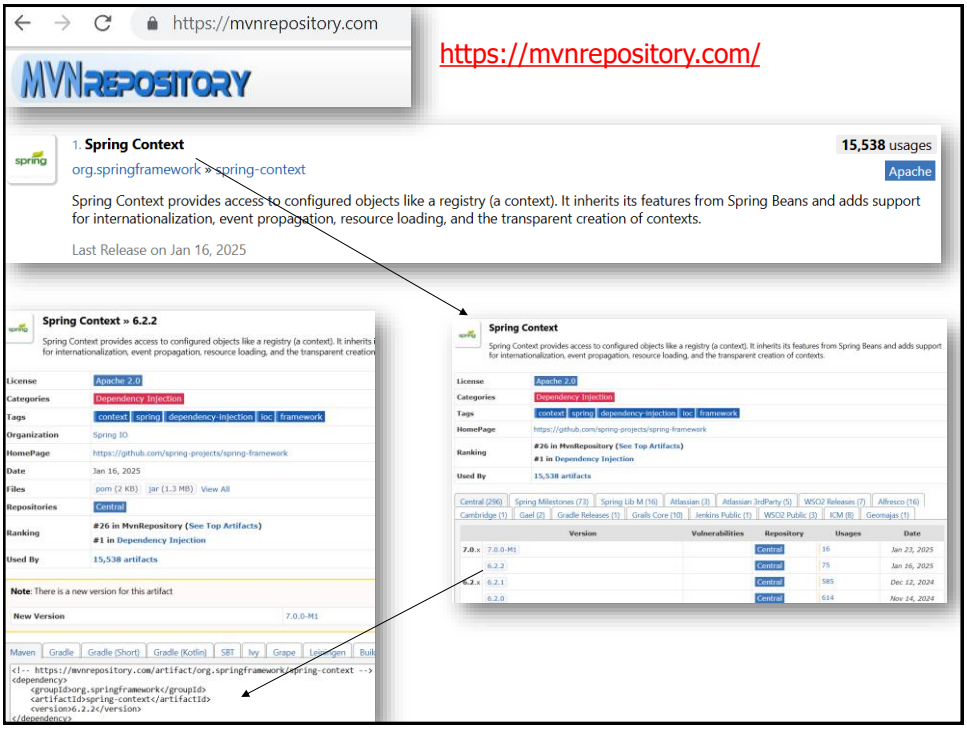
2.2 Example: wiring in Spring (IoC + field injection)

Maven Project

Spring_mvn_FirstExample [boot]

15

https://mvnrepository.com/



16

pom.xml

```xml
<!-- Spring -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>6.2.2</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>6.2.2</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>3.4.3</version>
</dependency>
```

17

Non-IoC style

```
public class Calculate
{
    public static void main(String args[]) {
        long op1 = Long.parseLong(args[0]);
        long op2 = Long.parseLong(args[1]);
        showResult("The result of %d %s %d is %d !".formatted(
                op1, getOpsName(),op2, operate(op1, op2))); }

        private static void showResult(String result) {
                System.out.println(result);}

        private static long operate(long op1, long op2){
                return op1 + op2; }

        private static String getOpsName(){
                return " plus " ; }
}
```

18

18

Non-IoC style

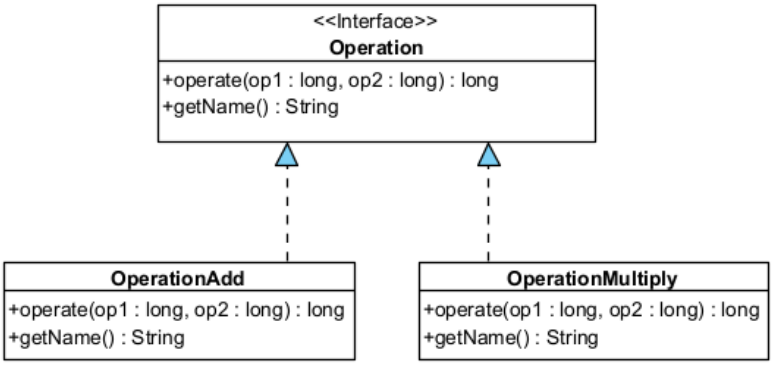
If you need to perform **multiplication instead of addition** on the operation,
the code must be **changed**.

If you need to write the result to **a file instead of the screen**,
the code must be **changed** again.

19

19

Creating a Modularized Application



We split up into an interface and implementation.

20

20

Creating a Modularized Application

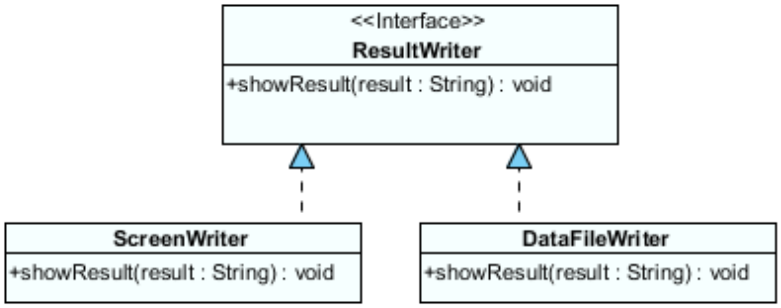
```
package domain;
public interface Operation {

        public long operate(long op1, long op2);
        public String getName();
}
```

```
package domain;
public class OperationAdd implements Operation{

    @Override
    public long operate(long op1, long op2) {
        return op1 + op2;
    }

    @Override
    public String getName() {
        return " plus ";
    }
}
```

21

21

Creating a Modularized Application



22

22

Creating a Modularized Application

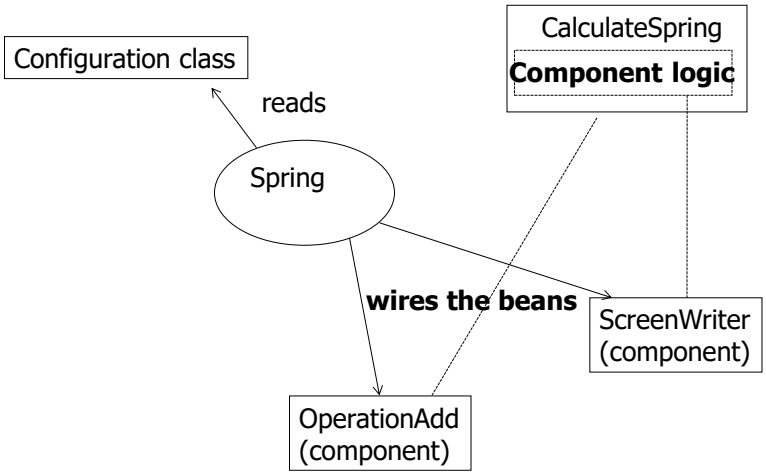
```
package domain;
public interface ResultWriter {

    public void showResult(String result);
}
```

```
package domain;

public class ScreenWriter implements ResultWriter{

        @Override
        public void showResult(String result) {
                System.out.println(result);
        }

}
```

23

23

Using Spring to Configure a Modularized Application



The **Spring container reads** the `configuration class`, instaniates the beans, and then **wires** them up according to the configuration information.

24

24

Using Spring to Configure a Modularized Application

Spring framework:
easily wire and
rewire reusable Java beans

Task:
instantiating concrete instances of `Operation` or `ResultWriter`

Class `CalculateSpring:`
delegates ***this task*** to the **Spring container**.

25

25

```
package spring_wiring;

import org.springframework.beans.factory.annotation.Autowired;

import domain.Operation;
import domain.ResultWriter;
import lombok.Getter;

@Getter
public class CalculateSpring {

    @Autowired
    private Operation ops;

    @Autowired
    private ResultWriter writer;
```

FIELD INJECTION

26

26

```
public void execute(String [] args)  {
        long op1 = Long.parseLong(args[0]);
        long op2 = Long.parseLong(args[1]);
        writer.showResult("The result of %s%s%s is %s!".
                formatted(op1, ops.getName(),
                        op2, ops.operate(op1, op2)));
  }

}
```

27

27

```
package main;
import configuration.FirstExampleConfiguration;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import spring_wiring.CalculateSpring;

public class StartUp {

    public static void main(String... args) {

        try (var context = new AnnotationConfigApplicationContext(
                                        FirstExampleConfiguration.class)) {

            var opsbean = context.getBean("opsbean", CalculateSpring.class);

            opsbean.execute(args);
        }
    }

}
```

28

28

The Bean Factory

The IoC container in Spring is called the **bean factory**.

bean factory = interface

will load bean definitions stored in a configuration source (such as a configuration class)

factory design pattern

```
ApplicationContext context
= new AnnotationConfigApplicationContext(FirstExampleConfiguration.class);
```

ApplicationContext is a subinterface of **BeanFactory** and adds **additional facilities** such as

resource bundles, integration with Spring AOP,
application event handling,
provide a generic way to load file resources, such as images, and more.

29

29

The bean factory reads the bean definitions from the configuration class.

It doesn't instantiate the beans.
Beans are **"lazily" loaded** into bean factories.

The beans will not be instantiated until they are needed.

`AnnotationConfigApplicationContext`

reads bean definitions from configuration classes.

30

30

```
public class CalculateSpring {

    @Autowired
    private Operation ops;

    @Autowired
    private ResultWriter writer;
    ...
```

CalculateSpring calculateSpring =
context.getBean("**opsbean**", CalculateSpring.class);

Configuration class:

```
@Bean
Operation operation() {
    return new OperationAdd();
}

@Bean
ResultWriter resultWriter() {
    return new ScreenWriter();
}

@Bean
CalculateSpring opsbean() {
    return new CalculateSpring(); }
```

31

31

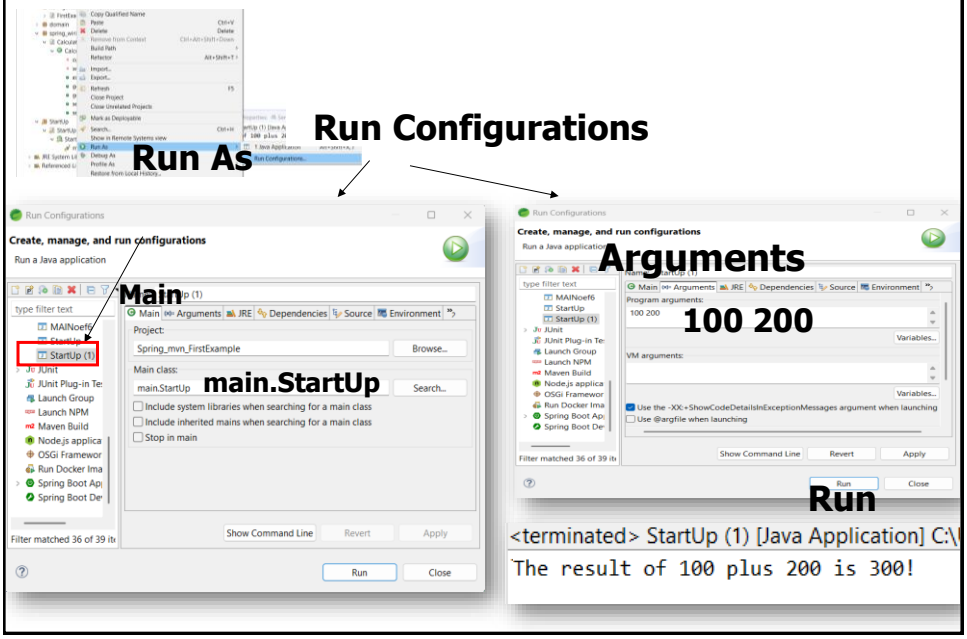
Configuration class:

```
...
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import spring_wiring.CalculateSpring;

@Configuration
public class FirstExampleConfiguration {

    @Bean
    ResultWriter resultWriter() {
        return new ScreenWriter();
    }

    @Bean
    Operation operation() {
        return new OperationAdd();
    }

    @Bean
    CalculateSpring opsbean() {
        return new CalculateSpring();
    }
}
```

32

32

1. Run as → Java Application
2. Run as → Run Configurations →

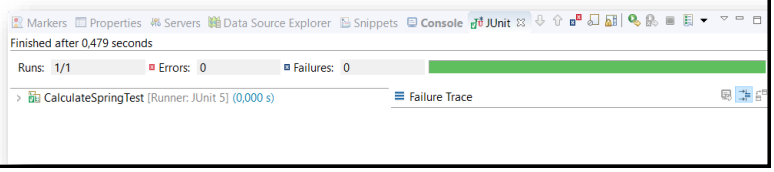


33

```
package test;

import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.junit.jupiter.web.SpringJUnitWebConfig;

import configuration.FirstExampleConfiguration;
import spring_wiring.CalculateSpring;

@SpringJUnitWebConfig(FirstExampleConfiguration.class)
class CalculateSpringTest {
```

34

```
@Autowired
private CalculateSpring calculateSpring;

@Test
void test()
{
    Operation op = calculateSpring.getOps();
    assertInstanceOf(OperationAdd.class, op);
    assertInstanceOf(ScreenWriter.class, calculateSpring.getWriter());
    assertEquals(300, op.operate(100, 200));
}

}
```



35