Spring

> *"Spring helps development teams everywhere to build simple, portable, fast and flexible JVM-based systems and applications."*

Why Spring?

* Writing nicer object oriented software
* Enterprise ready
  * Transactions
  * Security
* Easy to use
* Alternative for Java EE

Agenda

- Spring introduction
- Dependency injection
- Spring Boot
- In memory database
- Data access
- Transactions
- JPA

Agenda continued:

- MVC
- REST
- AOP
- External values
- Caching
- Asynchronous
- Scheduling
- JMS
- Security
- Conclusion

Rules of engagement

- Training hours
- Lunches
- Phones
- Training material
- Evaluation
- It's your course!

Have fun!

Spring introduction

History

- 2003: Rod Johnson released a Java EE book. The book addresses some problems with Java EE and a framework to solve those issues: Spring. It's mainly aimed as a means for inversion of control (dependency injection)
- 2004: Spring release 1.0
- 2009: Spring becomes part of VMWare
- 2013: Spring was transferred to Pivotal Software a joint venture between VMWare and EMC Corporation
- 2014: first release of Spring Boot

The Spring Platform

- Is a collection of some 20 up modules
- Modules are grouped along technologies:
  - Core Container
  - Data Access / integration
  - Dependency injection
  - AOP (Aspect Oriented Programming)
  - Transaction management
  - Security
  - Messaging
  - Test

Spring Tool Suite (STS)

- Eclipse based IDE
- Possible to install as a plugin in Eclipse or standalone
- You can pick whatever IDE you want

Spring vs Java EE

- Spring can be seen as an alternative to Java EE
- Although Spring is proprietary, the development of the platform seems uninterrupted
- Spring is evolving faster

Dependency injection

Why dependency injection?

- To follow object oriented principles
  - Single responsibility principle
  - Decoupling
  - Makes it easier to manage dependencies between objects
  - Software with dependency injection is easier to test

Plain Java example

```
public class UserController {
    private UserRepository userRepository = new UserRepository();
}
```

- The UserController needs to 'know' (instantiate) the UserRepository
- It's not the UserController's responsibility from an object oriented perspective

Plain Java example

```
public class UserController {
    private UserRepository userRepository = new UserRepository();
}
```

Spring dependency injection example

```
public class UserController {
    @Autowired
    private UserRepository userRepository;
}
```

Each Spring application has an ApplicationContext

- The ApplicationContext is also called the Spring container
- It is an Inversion of Control (IoC) container taking care of Dependency Injection (DI)
- Configuration metadata can be supplied in 3 ways
  1. Defined in a XML
  2. Defined with Java code
  3. Defined with annotations

Configuration metadata

- Our application consists of:
  - Business objects (POJO's)
  - Configuration metadata
- The Spring container is created based on those
- The Spring container produces a running application

POJO

> *"a plain old Java object (POJO) is an ordinary Java object, not bound by any special restriction and not requiring any class path"*

- Spring bean is a POJO, optionally implements an interface
- Must have a no arg-constructor

Example POJO

```
public class POJO {
  private String property;

  public String getProperty() {
    return property;
  }

  public void setProperty(String property) {
    this.property = property;
  }
}
```

XML configuration

The ApplicationContext

- Defined in a XML file for instance application-config.xml
- Should be on the classpath (in src/main/resources) of the application

```
<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" >

<!--Configuration -->

</beans>
```

Spring beans

- Any Java class configured in applicationContext.xml
- A Spring bean is singleton by default

```
<bean id="printer" class="lab1.ConsolePrinter" />
```

- *printer* is the unique identifier
- *lab1.ConsolePrinter* is the fully qualified classname

Spring bean example

- Plain POJO class
  - optionally implement an interface
- Must have no-arg constructor

```
public class ConsolePrinter implements PrinterService {
  public void print(String message) {
    System.out.println(message);
  }
}
```

Starting a Spring container

- Just for stand-alone applications

```
ApplicationContext applicationContext =
new ClassPathXmlApplicationContext("application-config.xml");

// If ConsolePrinter has a unique implementation.

ConsolePrinter printer =
applicationContext.getBean(ConsolePrinter.class);

// Retrieves the "printer" bean of the type ConsolePrinter.

ConsolePrinter printer2 =
applicationContext.getBean("printer", ConsolePrinter.class);

// Retrieves the "printer" bean, needs a type cast.

ConsolePrinter printer3 =
(ConsolePrinter) applicationContext.getBean("printer");
```

Dependency Injection

- Only define a dependency, don’t instantiate or lookup dependencies
- Spring wires dependencies
  - Constructor injection
  - Setter injection
- Dependencies are injected at bean creation time

Constructor Injection

```
public class ConstructorDI {
  private PrinterService printerService;

  public ConstructorDI(PrinterService printerService) {
    this.printerService = printerService;
  }
}
```

```
<bean id="printerService" class="lab1.ConsolePrinter" />

<bean id="constructorDI" class="lab1.ConstructorDI">
<constructor-arg ref="printerService" />
</bean>
```

Constructor Injection

- Arguments are resolved by type
  - argument order cannot be determined

```
public ConstructorDI(PrinterService printerService,
                     OtherDependency otherDependency) {
    this.printerService = printerService;
    this.otherDependency = otherDependency;
}

<bean id="printerService" class="lab1.ConsolePrinter" />

<bean id="constructorDI" class="lab1.ConstructorDI">
<constructor-arg ref="printerService"/>
<constructor-arg ref="otherDependency"/>
</bean>
```

Constructor type ambiguity

- Remove ambiguity using explicit types.

```
public ConstructorDI(String myString, int myInt) {
   this.myString = myString;
   this.myInt = myInt;
}
```

```
<bean id="constructorDI" class="lab1.ConstructorDI">
<constructor-arg type="java.lang.String" value="Hello" />
<constructor-arg type="int" value="1" />
</bean>
```

Constructor type ambiguity

- Remove ambiguity using explicit types.

```
public ConstructorDI(String myString, int myInt) {
   this.myString = myString;
   this.myInt = myInt;
}
```

```
<bean id="constructorDI" class="lab1.ConstructorDI">
<constructor-arg index="0" value="Hello" />
<constructor-arg index="1" value="1" />
</bean>
```

Setter injection

```
public class SetterDI {

  private PrinterService printerService;

  public void setPrinterService(PrinterService printerService) {
    this.printerService = printerService;
  }
}
```

```
<bean id="printerService" class="lab1.ConsolePrinter" />

<bean id="setterDI" class="lab1.SetterDI" />
<property name="printerService" ref="printerService"/>
</bean>
```

AutoWiring

- Wire dependencies automatically
- Don't specify dependencies using <property> or <constructor-arg/>
    - Less XML configuration
    - Easier to add or remove dependencies
    - Can be turned on globally or per bean

AutoWiring Types

Mode	**Explanation**
no	No autowiring (default)
byName	Property name must match bean id
byType	Property type must match **exactly one** bean with a default constructor.
constructor	Injection by type using a constructor. All arguments must be resolvable.

Autowiring byType

```
<bean id="printerService" class="lab1.ConsolePrinter"/>

<bean id="autoWireDI" class="lab1.AutoWireDI" autowire="byType"/>
```

```
public class AutoWireDI {

    private PrinterService printerService;

    public void setPrinter(PrinterService printerService) {
        this.printerService = printerService;
    }
}
```

Autowiring constructor

```
<bean id="printerService" class="lab1.ConsolePrinter"/>

<bean id="autoWireDI" class="lab1.AutoWireDI" autowire="constructor"/>
```

```
public class AutoWireDI {

    private PrinterService printerService;

    public AutoWireDI(PrinterService printerService) {
       this.printerService = printerService;
    }
}
```

Autowiring byName

```
<bean id="printerService" class="lab1.ConsolePrinter"/>

<bean id="autowiringDI" class="lab1.AutoWireDI" autowire="byName"/>
```

```
public class AutoWireDI {

    private PrinterService printerService;

    public void setPrinterService(PrinterService printerService) {
       this.printerService = printerService;
    }
}
```

- Bean id "printerService" should match "setPrinterService"

Global AutoWiring

```
<beans ... default-autowire="byType" />
```

Autowire-candidate

- When multiple implementations of an interface exist Spring can't autowire
- You have to choose which implementation to inject

  - Don't use for injection

```
<bean id="otherDependency" class="lab1.OtherDependency"
autowire-candidate="false"/>
```

  - Always use for injection

```
<bean id="otherDependency" class="lab1.OtherDependency"
primary="true"/>
```

Life-cycle callbacks

```
public class LifecycleCallbacks {
    public void init() {
      System.out.println("Creating bean");
    }

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

```
<bean id="lifecycleCallbacks" class="lab1.LifecycleCallbacks"
init-method="init" destroy-method="destroy"/>
```

Bean definition inheritance

- No instance is created of the abstract bean

```
<bean id="parentBean" abstract="true" class="lab1.ParentBean">
<property name="name" value="parent"/>
<property name="age" value="1"/>
</bean>

<bean id="inheritsWithDifferentName" parent="parentBean">
<property name="name" value="override"/>
</bean>
```

Java configuration

Spring beans

```
@Configuration
public class TestConfig {

    @Bean
    public PrinterService printerService(){
        return new ConsolePrinter();
    }
}
```

- This creates a Spring bean implementing the PrinterService interface.
- A Spring bean is a Singleton by default.

Getting a reference to a Spring bean

```
@Configuration
public class TestConfig {

    @Bean
    public PrinterService printerService(){
        return new ConsolePrinter();
    }
}
```

Getting a reference to a Spring bean continued

```
try (AnnotationConfigApplicationContext context =
      new AnnotationConfigApplicationContext(TestConfig.class)) {

    PrinterService printerService1 =
           context.getBean(PrinterService.class);
    printerService1.print("Hello context");

    PrinterService printerService2 =
           context.getBean("printerService",PrinterService.class);
    printerService2.print("Hello context");

    PrinterService printerService3 =
           (PrinterService)context.getBean("printerService");
    printerService3.print("Hello context");
}
```

Constructor Injection

```
public class ConstructorDI {

    private PrinterService printerService;

    public ConstructorDI(PrinterService printerService){
        this.printerService=printerService;
    }
}

@Configuration
public class TestConfig {

    @Bean
    public PrinterService printerService(){
        return new ConsolePrinter();
    }

    @Bean
    public ConstructorDI constructorDI(PrinterService printerService){
        return new ConstructorDI(printerService);
    }
}
```

Constructor Injection

- Arguments are resolved by type

```
@Configuration
public class TestConfig {
    @Bean
    public PrinterService printerService(){
      return new ConsolePrinter();
    }
    @Bean
    public OtherDependency otherDependency(){
      return new OtherDependency();
    }

@Bean
public ConstructorDI constructorDI(PrinterService printerService,
                                   OtherDependency otherDependency){
            return new ConstructorDI(printerService, otherDependency);
        }
}
```

Setter Injection

```
public class SetterDI {
  private PrinterService printerService;

  public void setPrinterService(PrinterService printerService) {
    this.printerService = printerService;
  }
}

@Configuration
public class TestConfig {
    @Bean
    public PrinterService printerService(){
      return new ConsolePrinter();
    }

    @Bean
    public SetterDI setterDI(PrinterService printerService){
      SetterDI setterDI = new SetterDI();
      setterDI.setPrinterService(printerService);
      return setterDI;
    }
}
```

AutoWiring

- Wire dependencies automatically
- Use the container to collect the dependencies

```
public class AutoWireDI {

    @Autowired
    private PrinterService printerService;

    public void sayHello(){
        printerService.print("Hello from autowired bean");
    }
}
```

Factory method

A Factory allows an object to create new objects without having to know the details. The object doesn't know how the other objects are created or what the dependencies of those objects are.

Factory Method

```
public class FactoryMethod{
    private PrinterService printer; private String value;

    public void printValue() {
       printer.print(value);
    }
    public static FactoryMethod create(PrinterService printer,
                                       String value) {
       FactoryMethod instance = new FactoryMethod();
       instance.printer = printer;
       instance.value = value;
       return instance;
    }
}
```

Factory Method cont.

```
@Bean
public PrinterService printer() {
   return new ConsolePrinter();
}

@Bean
public FactoryMethod factoryMethod(PrinterService printer) {
    return FactoryMethod.create(printer, "MyTest");
}
```

Factory bean

- Construct and initialize beans that you don't control

```
public class NameFactory {

  public List<String> createNameList() {
    return Arrays.asList("Jan", "Alie", "Bert");
  }
}

@Configuration
public class TestConfig {

    @Bean
    public NameFactory nameFactory() {
         return new NameFactory();
    }
    @Bean
    public List<String> names() {
        return nameFactory().createNameList();
    }
```

Scopes

Scope	Explanation
singleton (default)	A single instance of a bean. Dependencies to the bean are shared.
prototype	New instance is created once for each injection point.
prototype + method injection	New instance is created for each call to the injected method.
request (web only)	Instance per HTTP request.
session (web only)	Instance per HTTP session.

Singleton Scope (default)

- There is **only one** HitCounter instance for all hitter's

```
@Bean
@Scope(scopeName="singleton")
public HitCounter hitCounter(){
    return new HitCounter();
}

@Bean
public Hitter hitter1(HitCounter hitCounter){
    return new Hitter( hitCounter);
}

@Bean
public Hitter hitter2(HitCounter hitCounter){
    return new Hitter( hitCounter);
}
```

Prototype Scope

- There is **one** HitCounter instance for every hitter

```
@Bean
@Scope(scopeName="prototype")
public HitCounter hitCounter(){
    return new HitCounter();
}

@Bean
public Hitter hitter1(HitCounter hitCounter){
    return new Hitter( hitCounter);
}
@Bean
public Hitter hitter2(HitCounter hitCounter){
    return new Hitter( hitCounter);
}
```

The Bean annotation

```
public @interface Bean {

    String[] name() default {};

    Autowire autowire() default Autowire.NO;

    String initMethod() default "";

    String destroyMethod() default AbstractBeanDefinition.INFER_METHOD;

}
```

- String[] name() => aliases for this bean
- String initMethod() => method to call during initialization
- String destroyMethod() => method to call upon closing the application context

Annotation configuration

Defining Spring beans

```
@Component
public class ConsolePrinter implements PrinterService {

    @Override
    public void print(String message) {
        System.out.println("Message: " + message);
    }
}
```

Is equivalent to:

```
@Bean
public PrinterService consolePrinter() {
    return new ConsolePrinter();
}
```

Instruct the Spring container to look for @Component

```
try (AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext()) {

    context.scan("com.example");
    context.refresh();

    PrinterService bean1 = context.getBean(PrinterService.class);
    bean1.print("Hello context");

    PrinterService bean2 =
           context.getBean("consolePrinter", PrinterService.class);
    bean2.print("Hello context");

    PrinterService bean3 =
                (PrinterService) context.getBean("consolePrinter");
    bean3.print("Hello context");
}
```

Instruct the Spring container to look for @Component alternative

```
@Configuration
@ComponentScan(basePackages="com.example")
public class ApplicationConfiguration {

}
```

@Autowired

```
// Field injection

@Autowired
private PrinterService printerService;

// Constructor injection

@Autowired
public ConstructorDI(PrinterService printerService) {
    this.printerService = printerService;
}
// Setter injection

@Autowired
public void setPrinterService(PrinterService printerService) {
    this.printerService = printerService;
}
```

Logging by the container

```
Returning cached instance of singleton bean 'consolePrinter'
Autowiring by type bean 'setterDI' via constructor to  'consolePrinter'
```

Method injection

- Normally injection is done at bean creation time
- Injecting a prototype scope bean in singleton scope bean effectively makes the prototype scope bean a singleton scope bean
- Method injection allows injection at each call to a bean
- Spring injects an abstract method implementation

Method injection

```
public class MySingleton {
@Autowired
    private javax.inject.Provider<MyPrototype> myPrototype;

    public void something(){
        MyPrototypeBean instance = myPrototype.get();
    }
}
```

Method injection alternative

```
public class MySingleton {

     public void something(){
         MyProtoTypeBean instance = getPrototypeBean();
     }

    @Lookup
    public MyProtoTypeBean getPrototypeBean() {
    // Spring will override this method
        return null;
    }
}
```

Lifecycle callbacks

(not Spring specific)

```
@SpringBootApplication
public class LifecycleApplication implements CommandLineRunner {
    ...
    @Override
    public void run(String... arg0) throws Exception {
    System.out.println("Run");
    }
    @PostConstruct
    public void init() {
    System.out.println("PostConstruct");
    }
    @PreDestroy
    public void clean() {
    System.out.println("PreDestroy");
}
```

```
PostConstruct
Run
PreDestroy
```

Conforming to standards javax.inject.Inject

- Same semantics as @Autowired
- Standardized in JSR-330
- Needs JSR-330 on the classpath

```
@Inject
private PrinterService printerService;
```

```
<dependency>
    <groupid>javax.inject</groupid>
    <artifactid>javax.inject</artifactid>
    <version>[version]</version>
</dependency>
```

@Autowired versus @Inject

About @Scope

> *"The JSR-330 default scope is like Spring's prototype. However, to keep it consistent with Spring's general defaults, JSR-330 bean declared in the container is a singleton by default. Use Spring's @Scope to use other scopes. @Scope is also defined in*

> *javax.inject but is used for creating your own annotations."*

Qualifiers

- Qualifiers bind an injection point to a specific implementation of an interface
  - runtime binding, not compile time
  - unit testing still possible
- Necessary when multiple implementations exist

String based Qualifiers

- Use Qualifier when declaring spring bean

```
@Component
@Qualifier("file")
public class FilePrinter implements PrinterServ

  @Override
  public void print(String message) {...}
}
//Using Qualifier

 @Autowired
 @Qualifier("file")
 private PrinterService printerService;
```

Custom Qualifier annotation

```
@Retention(RUNTIME)
@Target(value={FIELD,TYPE,CONSTRUCTOR,METHOD})
@Qualifier
public @interface File {
}

@Component
@File
public class FilePrinter implements PrinterService{

@Autowired
@File
private PrinterService printerService;
```

Injecting multiple implementations

- All MovieCatalog implementations

```
@Autowired
private List<PrinterService> services;
```

- All MovieCatalog implementations with bean id

```
@Autowired
private Map<String,PrinterService> services;
```

Importing configuration

```
@Configuration
@Import({ DatasourceConfig.class, ControllerConfig.class })
public class ApplicationConfig {

}
```

Dependency injection summary

- Only define a dependency, don't instantiate or lookup dependencies
- Spring wires dependencies
  - Constructor injection
  - Setter injection
  - Field injection
- Dependencies are injected at bean creation time

Best form of dependency injection

Constructor injection

- For mandatory dependencies
- To create immutable objects by assigning dependencies to final fields

Setter injection

- For optional dependencies.

Field injection

- Annotations are clearly visible when opening the class file
- Uses reflection
- Classes are tightly coupled to dependency injection and cannot be used without it
- Harder to unit test
- Dependencies are hidden from the outside
- Unable to create immutable objects
- Considered a bad practice in most cases

From the Spring docs 1.

> *The Spring team generally advocates constructor injection as it enables one to implement application components as immutable objects and to ensure that required dependencies are not null. Furthermore constructor-injected components are always returned to client (calling) code in a fully initialized state.*

From the Spring docs 2.

> *As a side note, a large number of constructor arguments is a bad code smell, implying that the class likely has too many responsibilities and should be refactored to better address proper separation of concerns.*

From Spring doc 3.

> *Setter injection should primarily only be used for optional dependencies that can be assigned reasonable default values within the class. Otherwise, not-null checks must be performed everywhere the code uses the dependency.*

From Spring doc 3.

> *One benefit of setter injection is that setter methods make objects of that class amenable to reconfiguration or re-injection later. Management through JMX MBeans is therefore a compelling use case for setter injection.*