



Spring Essentials

David THIBAU – 2022

david.thibau@gmail.com



Agenda

- **Introduction**
 - A brief history
 - IoC and Dependency Injection
 - Framework evolutions
- **Spring Core**
 - The IoC container
 - BeanFactory and ApplicationContext
 - Beans
 - Other core features
- **Annotations**
 - Configuration Classes
 - @Component and stereotypes
 - Dependency Injection
 - Environment and profiles
- **Spring AOP**
 - AOP concepts
 - Spring AOP features
 - AOP with XML
 - AOP with @AspectJ
- **Spring Boot**
 - The main concept of auto-configuration
 - Starters
 - Configuration properties with SpringBoot
- **Spring and persistence**
 - Spring Data
 - JPA
 - MongoDB
- **Web applications**
 - Spring MVC
 - Spring MVC for RestFul APIs
 - Serialization with Jackson
 - Exception handling, CORS, SpringDoc
- **Spring Security**
 - Spring Security
 - Stateful and stateless models
 - Autoconfiguration SpringBoot
 - oAuth2



Agenda

- **Testing with Spring**
 - Spring Test
 - Testing with SpringBoot
 - Auto-configured tests
- **Messaging with Spring**
 - Spring support for message brokers
- **Reactive Spring**
 - Reactive programming
 - The Reactor project
 - Spring Data reactive
 - Spring WebFlux
- **Towards production**
 - Actuator
 - Deployment



Introduction

A brief history

IoC and Dependency Injection
Framework evolutions



Spring companies

- ❖ Spring is an **OpenSource** project with a business model based on services (Support, Consulting, Training, Partnership and certifications)
- ❖ The **SpringSource**¹ company founded by the creators of Spring Rod Johnson and Juergen Hoeller was acquired by **VmWare**, then integrated into the joint venture **Pivotal Software**



Spring and Java EE

- ❖ Spring was initiated in opposition to the Java EE specification
- ❖ The founders of the solution have always claimed that Spring brings all the services of a Java EE server without the heavy specification.
 - Hence the name lightweight container
- ❖ After Oracle quit the Java EE specification, Spring is the most widely used Java framework for building back-end applications accessible via HTTP



The 7 commandments

- ❖ The 7 Commandments of Spring
 - **JavaEE is too complex**, it should be easier to use
 - It is always more advantageous to use **interfaces** rather than classes. Spring makes it easy to use interfaces.
 - **"Good" object programming** is far more important than implementing technologies like J2EE.
 - **Checked exceptions are used too much** in Java. A framework should not force developers to handle exceptions that cannot be controlled.
 - The **testability** of a program is essential. Spring should allow easy-to-test code.
 - Application code should **not be framework dependent**
 - Spring should not compete with good existing solutions but rather promote their **integration** (AspectJ, JDO, TopLink, Hibernate)
- ❖ Reference Book : *Expert One-on-One J2EE Design and Development* (Rod Johnson).



Benefits of Spring eco-system

- ❖ **Readable and simple** business tiers (Spring do the « plumbing »).
- ❖ A **unified format** for configuring all application layers
- ❖ Adoption of good programming practices in particular **interfaces**.
- ❖ **Dependency** on as few APIs as possible.
- ❖ **Testability**: Spring makes isolation of tested code easy.
- ❖ Declarative **technical services** (Transaction management, security for example).
- ❖ **Evolvutivity** : Spring's abstraction layers make it easy to switch from one technology to another (JDBC to JPA example)
- ❖ Use of **POJOs (Plain Old Java Objects)**.

What can Spring do?



Microservices

Quickly deliver production-grade features with independently evolvable microservices.



Reactive

Spring's asynchronous, nonblocking architecture means you can get more from your computing resources.



Cloud

Your code, any cloud—we've got you covered. Connect and scale your services, whatever your platform.



Web apps

Frameworks for fast, secure, and responsive web applications connected to any data store.



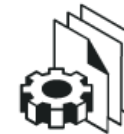
Serverless

The ultimate flexibility. Scale up on demand and scale to zero when there's no demand.



Event Driven

Integrate with your enterprise. React to business events. Act on your streaming data in realtime.



Batch

Automated tasks. Offline processing of data at a time to suit you.



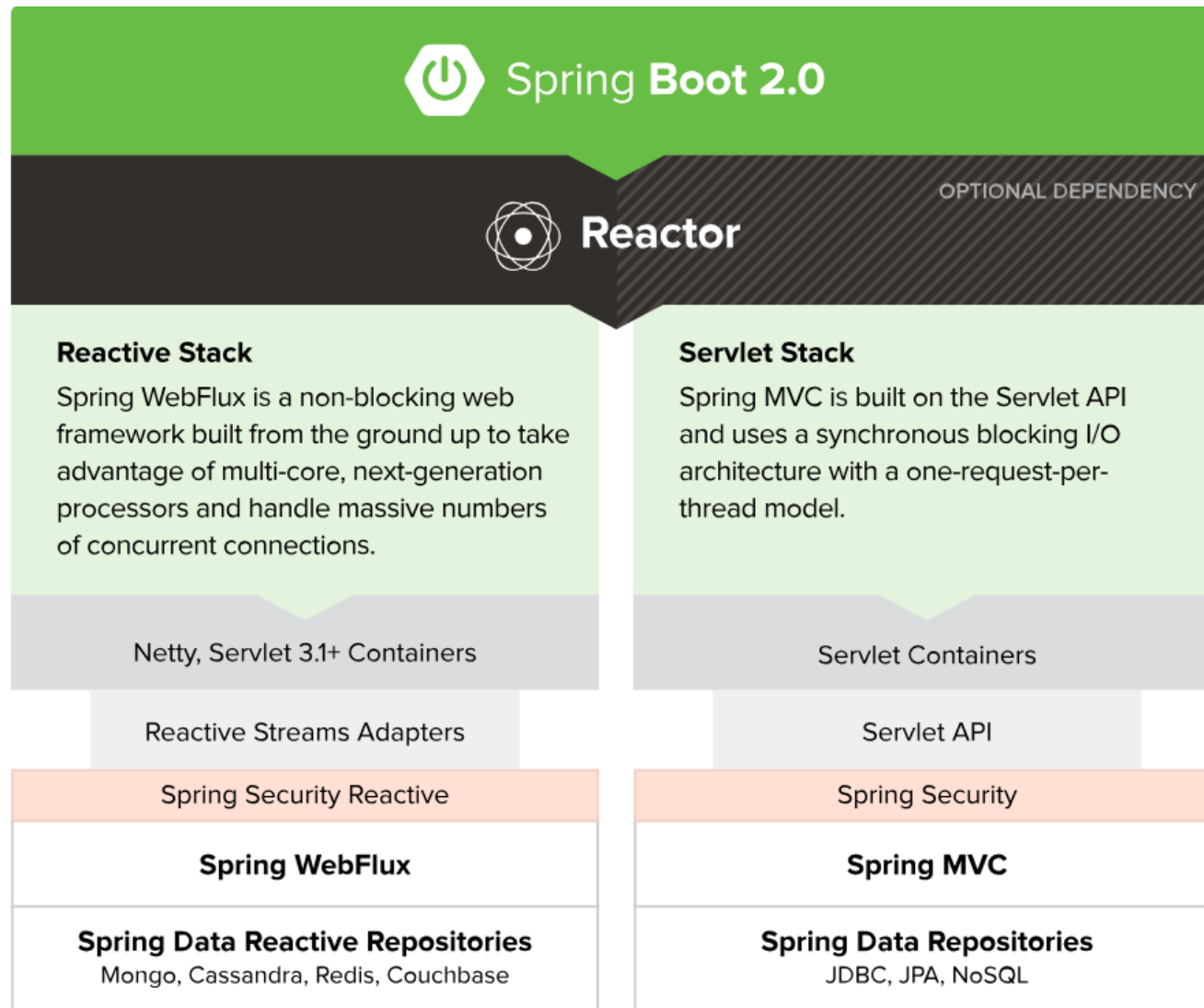
Spring Projects

Spring is a set of projects adapted to all development issues.

All these projects share Spring Core which allow :

- ✓ IoC and Dependency Injection
- ✓ AOP programming
- ✓ Configurability
- ✓ Testability

Web Stacks





Core modules

Container :

- Beans definitions : Instantiation and Dependency Injection
- *ApplicationContext* : Beans access, Resources Loading, Validation and Data Binding
- Expression Language : Language for referencing beans

AOP, Aspects et Instrumentation : Interceptors, cross-cutting concern, technical services

Test : JUnit and TestNG integration (JUnit5 Extensions)



Introduction

A brief history
IoC and Dependency Injection
Framework evolutions



IoC Pattern

❖ The problem :

How to make the web architecture based on controllers work with the interface to the database when these are developed by different teams?

❖ OOP response :

Of course, Use interfaces !!



Example

- ❖ We want to implement a business function allowing to list all the films of a given director regardless of the location where the movies are stored
- ❖ The *MovieController* class can rely on a *finder* object able to retrieve all the movies of a persistent store :

```
class MovieController...  
    public List<Movie> moviesDirectedBy(String arg) {  
        return finder.findAll().stream()  
            .filter( m -> m.getDirector().equals(arg) )  
            .collect(Collectors.toList());  
    }
```

- ❖ And as we want to be independent of the way movies are stored, we have defined an interface :

```
public interface MovieFinder {  
    List<Movie> findAll();  
}
```



Instanciación de un implementación ?

- ❖ Even if the code is well decoupled, to execute the code, we need a concrete class that implements the finder interface. For example, we can instantiate the concrete class in the constructor of *MovieController*.

```
class MovieController...  
    private MovieFinder finder;  
    public MovieController() {  
        finder = new ColonDelimitedMovieFinder("movies1.txt");  
    }
```

- ❖ Unfortunately, by doing that way, we loose all the benefits of the decoupling. Our controller is now dependent of a specific implementation !!
=> Solution : The **IoC pattern**



IoC and framework

- ❖ The ***IoC (Inversion Of Control)*** pattern means that it is no longer the developer's code that has execution control but the framework
- ❖ The framework is then responsible for instantiating objects, calling methods, freeing objects, etc.
- ❖ Developer code is limited to business code. It depends exclusively on interfaces and therefore be more scalable



IoC vs Dependency Injection

- ❖ **Dependency injection** is just a specialization of the IoC pattern
- ❖ The framework calls the methods to initialize the attributes of your objects.
- ❖ In the preceding illustration, it initializes the interface variable with an implementation object.



How to inject ?

❖ 3 main ways to inject dependencies:

- **Constructor injection** : Attributes are initialized in the constructor
- **Setter Injection** : Attributes are initialized via setter methods
- **Method injection** : Attributes are initialized via a specific méthode



Constructor Injection

- ❖ The *MovieController* class declare a constructor with required dependencies.

```
class MovieController...  
public MovieController(MovieFinder finder) {  
    this.finder = finder;  
}
```

- ❖ The *finder* object will also be instantiated by the framework.
ColonMovieFinder for example will provide a constructor allowing to set the filename's location

```
class ColonMovieFinder...  
public ColonMovieFinder(String filename) {  
    this.filename = filename;  
}
```



Setter Injection

```
class MovieLister...  
    private MovieFinder finder;  
    public void setFinder(MovieFinder finder) {  
        this.finder = finder;  
    }
```

```
class ColonMovieFinder...  
    public void setFilename(String filename) {  
        this.filename = filename;  
    }
```



Method Injection¹

```
public interface InjectFinder {  
    void injectFinder(MovieFinder finder);  
}  
class MovieLister implements InjectFinder...  
public void injectFinder(MovieFinder finder) {  
    this.finder = finder;  
}
```

```
public interface InjectFinderFilename {  
    void injectFilename (String filename);  
}  
class ColonMovieFinder implements MovieFinder,  
    InjectFinderFilename.....  
    public void injectFilename(String filename) {  
        this.filename = filename;  
    }
```

1. From Java5 and annotations, it is no longer necessary to define a particular interface and any annotated method can be used for injection



Framework configuration

- ❖ Configuration of the framework consist mainly to define implementations to use (for a particular run)
- ❖ Configuration is typically made
 - Via XML files
 - Or specific Java class associated with Java annotations



XML Configuration

```
<beans>
```

```
  <bean id="MovieLister" class="spring.MovieLister">
```

```
    <property name="finder" ref="MovieFinder"/>
```

```
  </bean>
```

```
  <bean id="MovieFinder" class="spring.ColonMovieFinder">
```

```
    <property name="filename">
```

```
      <value>movies1.txt</value>
```

```
    </property>
```

```
  </bean>
```

```
</beans>
```

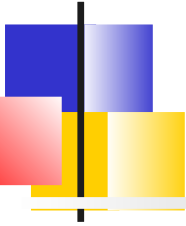



Loading config

```
public void testWithSpring()
    throws Exception {

    // Loading framework's configuration from XML
    ApplicationContext ctx =
        new FileSystemXmlApplicationContext("spring.xml");

    // Use of the instantiated bean
    MovieLister lister = (MovieLister)ctx.getBean("MovieLister");
    Movie[] movies = lister.moviesDirectedBy("Sergio Leone");
    assertEquals("Once Upon a Time in the West",movies[0].getTitle());
}
```



Benefits of dependency injection

- ❖ Application components are **easier to write**
- ❖ Components are **easier to test**. In test configuration, dependencies can be easily mock.
- ❖ The **typing of objects is preserved**.
- ❖ Dependencies are **self-explanatory**
- ❖ With XML config, most business objects **do not depend on the container API** and can therefore be used with or without the container.



In practice

With Spring a developer can :

- Write a method running in a database transaction without using the transaction API
- Make a method remotely accessible without using a remote API
- Defining a message handler method without using the message broker API
- ...



Introduction

A brief history
IoC and Dependency Injection
Framework evolution



Versions of Spring Framework

- ❖ 1.2 March 2004 : IoC container, AOP, MVC, Configuration XML, DAO : TopLink, JDO 2.0, Hibernate 3.0.3 support
- ❖ 2.0 October 2006 : XML namespaces, AOP with AspectJ , Specific Scopes
- ❖ 2.5 November 2007 : Java 6 Support, alignment with Java EE 5, Configuration via Annotations, JUnit4 and TestNG
- ❖ 3.0 Décembre 2009 : Java 5 minimum, Spring Expression Language, javax.Validation, Alignment with Java EE6, JavaConfig, Embedded Database support
- ❖ 4.0 December 2013 : Java 6 minimum, Support for Java8, Alignment with Java EE7, Groovy, Websocket
- ❖ 4.3 Juin 2016, Ease of configuration
- ❖ 5.0 2017 : Java 8+, Reactive programming with *Reactor*, *Kotlin* Support 1.1+, JUnit 5, Java EE 8 API level



Versions of SpringBoot

SpringBoot reduce the effort for configuring the framework

April 2014, spring boot 1.x for Spring4

March 2018, spring boot 2.x for Spring5



Configuration

❖ 3 choices to configure the container :

- **XML :**

- Modification without recompilation
- Use of specific *namespaces*
- Present in documentation reference

- **Java :**

- Configuration classes
- Annotation for beans and dependency injection

- **Spring Boot + Java :**

- Deduced default configuration + annotations
- The modern way



XML Configuration

```
<beans>
```

```
  <bean id="MovieLister" class="spring.MovieLister">
```

```
    <property name="finder">
```

```
      <ref local="MovieFinder"/>
```

```
    </property>
```

```
  </bean>
```

```
  <bean id="MovieFinder" class="spring.ColonMovieFinder">
```

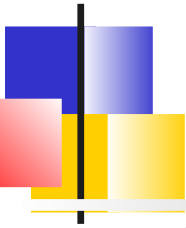
```
    <property name="filename">
```

```
      <value>movies1.txt</value>
```

```
    </property>
```

```
  </bean>
```

```
</beans>
```

Configuration via annotations

```
@Import(DataSourceConfig.class)  
@Configuration  
public class SimpleConfiguration {  
    @Autowired  
    Connection connection;  
  
    @Bean  
    Database getDatabaseConnection(){  
        return connection.getDBConnection();  
    }  
    // More code here....  
}
```



Bean definition and injection via annotations

@RestController

```
public class MovieController {  
    MovieFinder finder ;
```

@Autowired

```
public MovieController(MovieFinder finder) {  
    this.finder = finder ;  
}  
  
public List<Movie> moviesDirectedBy(String arg) {  
    List<Movie> allMovies = finder.findAll();  
    List<Movie> ret = new ArrayList<Movie>() ;  
    for (Movie movie : allMovies ) {  
        if (!movie.getDirector().equals(arg))  
            ret.add(movie);  
    }  
    return ret;  
}
```



SpringCore

Spring : a lightweight container
BeanFactory and ApplicationContext
Beans
Other core features

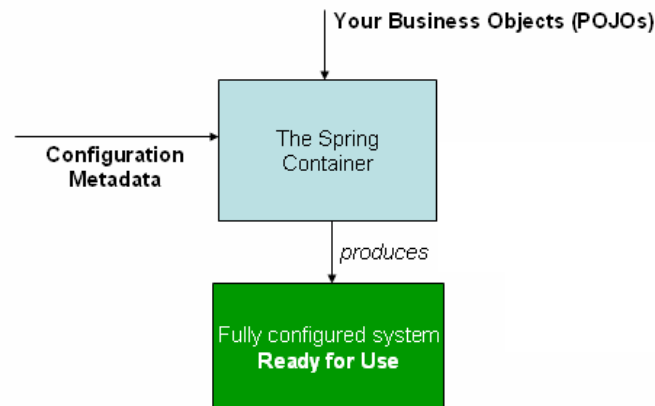


Boot process

Objects (beans) define their dependencies and provide injection methods .

The container reads its configuration, instantiates the beans and injects their dependencies

Beans are then ready to use.





Loading the config

Several ways to load the config :

- Read one or several XML files from the classpath or the file system
- Scan of packages and discover annotations (@Configuration, @Bean, ...)
- Read one or several properties, yml files to initialize some values of the beans (Basic types attributes).
- Groovy files
- Kotlin files
- ...



Use of the container

Loading the configuration produces an ***ApplicationContext*** object which contains the bean's definitions

ApplicationContext is the container

Main methods are :

```
String [] getBeanDefinitionNames()
```

```
T getBean(String name, Class<T> requiredType)
```



Examples

// Loading XMLs via the classpath

```
ApplicationContext context = new ClassPathXmlApplicationContext("services.xml",  
    "daos.xml");
```

// Get an instance

```
PetStoreService service = context.getBean("petStore", PetStoreService.class);
```

// Use of the bean

```
List<String> userList = service.getUsernameList();
```

-- Via Groovy

```
ApplicationContext context = new  
    GenericGroovyApplicationContext("services.groovy", "daos.groovy");
```

-- Via Kotlin

```
val context = GenericApplicationContext()  
GroovyBeanDefinitionReader(context).loadBeanDefinitions("services.groovy",  
    "daos.groovy")  
context.refresh()
```



SpringCore

Spring : a lightweight container
BeanFactory and ApplicationContext
Beans
Other core features



API

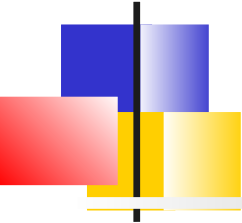
Packages ***org.springframework.beans*** and ***org.springframework.context*** are the base packages of the container.

The ***BeanFactory*** interface contains the configuration mechanisms.

ApplicationContext is a sub-interface of *BeanFactory* which adds enterprise features:

- Messages resolution
- Support of internationalization
- Publishing Events
- Application-layer specific contexts (for example. Web application contexts)

The *ApplicationContext* represents the container



BeanFactory vs ApplicationContext

Table 9. Feature Matrix

Feature	BeanFactory	ApplicationContext
Bean instantiation/wiring	Yes	Yes
Integrated lifecycle management	No	Yes
Automatic <code>BeanPostProcessor</code> registration	No	Yes
Automatic <code>BeanFactoryPostProcessor</code> registration	No	Yes
Convenient <code>MessageSource</code> access (for internationalization)	No	Yes
Built-in <code>ApplicationEvent</code> publication mechanism	No	Yes



Support of internationalization

- ❖ A bean named ***messageSource*** implementing the *MessageSource* interface is declared in the configuration. It is automatically detected when loading the *ApplicationContext*
- ❖ *getMessage()* methods are used to retrieve a localized message.
messageSource.getMessage("account.name", null, Locale.ENGLISH);
- ❖ The classic implementation of *MessageSource* is
org.springframework.context.support.ResourceBundleMessageSource



Example

```
<beans>
  <bean id="messageSource"
    class="org.springframework.context.support.ResourceBundleMessageSource">
    <property name="baseName" value="WEB-INF/test-messages"/>
  </bean>
  <bean id="example" class="com.foo.Example">
    <property name="messages" ref="messageSource"/>
  </bean>
</beans>

public class Example {
  private MessageSource messages;

  public void setMessages(MessageSource messages) {
    this.messages = messages;
  }
  public void execute() {
    String message = this.messages.getMessage("argument.required",
                                              new Object [] {"userDao"}, "Required", null);
    System.out.println(message);
  }
}
```



Event model

- ❖ Beans implementing the *ApplicationListener* interface receive events of type *ApplicationEvent*:
 - **ContextRefreshedEvent** : *ApplicationContext* is initialized or reloaded
 - **ContextClosedEvent** : *ApplicationContext* is closed and beans are destroyed
 - **RequestHandledEvent** : Specific to *WebApplicationContext*, an http request has been served.
- ❖ It is also possible to define custom events and Spring projects can use this event model

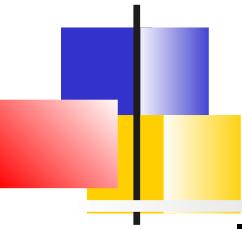


SpringCore

Spring : a lightweight container
BeanFactory and *ApplicationContext*

Beans

Other core features



BeanDefinition

Inside the container, beans are represented by their ***BeanDefinition*** class which encapsulates:

- The **qualified name** of the associated class
- The **behavioral configuration** of the bean (scope, call-back methods, ...).
- **Dependencies** : References to other beans
- **Other** configuration data (e.g. pool sizing)



Bean's names

A bean has at least one identifier, **its name** which is unique inside the container.

- The convention is to use the standard Java convention for instance attributes.

Other names can be defined, they are called ***aliases***.



Instantiation

BeanDefinition are used to instantiate the beans

The ***class*** property can be used in 2 different ways :

- Directly to instantiate the constructor via reflection
- To specify a factory class



Lifespan of beans

Beans are simple POJOS which may have 3 different lifespan (or scope) :

- **Singleton**: There is only one instance of the object (which is therefore shared). Ideal for stateless services
- **Prototype** : Each time the object is requested via its name, a new instance is created.
- **Custom object “scopes”** : Their life cycle is usually scoped onto another object.
Ex: The HTTP request, the session, a DB transaction



Aware interfaces

- ❖ Les beans may implement **Aware** interfaces which allow to interact with the container during the boot process:
 - **BeanNameAware** : To be notified with the bean's name
 - **ResourceLoaderAware, ApplicationContextAware, ApplicationEventPublisherAware** : To have reference of the resource loader, the event publisher or directly the container (Generally, it is always *ApplicationContext*)
 - **MessageSourceAware** : The messageSource
 - **ServletContextAware** : The Servlet Context



PostProcessor interfaces

2 hooks are provided in the construction process, they can be used to extend the behavior of the container :

- ***BeanPostProcessor*** : Custom modification of new bean instances. Typically used to apply aspects via annotations. (See AOP)
- ***BeanFactoryPostProcessor*** allows for custom modification of a bean definition



BeanPostProcessor

- ❖ 2 methods for *BeanPostProcessor*
 - ***Object postProcessBeforeInitialization(Object bean, String beanName)***
 - ***Object postProcessAfterInitialization(Object bean, String beanName)***
- ❖ Beans implementing *BeanPostProcessor* must be declared in the configuration as beans.
- ❖ In case of several *BeanPostProcessor*, an order attribute can be specified to control the sequence of the methods



BeanFactoryPostProcessor

- ❖ *BeanFactoryPostProcessor* has a single method:

`postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory)`

- ❖ A bean implementing this interface may modify the factory used to create the bean
- ❖ Spring offers several useful implementations like *PropertySourcesPlaceholderConfigurer*



Example :

PropertyPlaceholderConfigurer

```
<bean
  class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
  <property name="locations">
    <value>classpath:com/foo/jdbc.properties</value>
  </property>
</bean>
<bean id="dataSource" destroy-method="close"
class="org.apache.commons.dbcp.BasicDataSource">
  <property name="driverClassName" value="${jdbc.driverClassName}"/>
  <property name="url" value="${jdbc.url}"/>
  <property name="username" value="${jdbc.username}"/>
  <property name="password" value="${jdbc.password}"/>
</bean>
```

❖ File *jdbc.properties*

```
jdbc.driverClassName=org.hsqldb.jdbcDriver
jdbc.url=jdbc:hsqldb:hsql://production:9002
jdbc.username=sa
jdbc.password=root
```



Sequence of bean instantiation

- If the bean implements BeanNameAware, call **setBeanName()**
- If the bean implements BeanClassLoaderAware call **setBeanClassLoader()**
- If the bean implements BeanFactoryAware, call **setBeanFactory()**
- If the bean implements ResourceLoaderAware, call **setResourceLoader()**
- If the bean implements ApplicationEventPublisherAware, call **setApplicationEventPublisher()**
- If the bean implements MessageSourceAware, call **setMessageSource()**
- If the bean implements ApplicationContextAware call **setApplicationContext()**
- If the bean implements ServletContextAware, call **setServletContext()**
- Calling **postProcessBeforeInitialization()** methods of BeanPostProcessors
- If the bean implements InitializingBean, call **afterPropertiesSet()**
- If *init-method* is set, call the specific *init* method
- Calling **postProcessAfterInitialization()** methods of BeanPostProcessors



SpringCore

Spring : a lightweight container
BeanFactory and *ApplicationContext*
Beans

Other core features



Loading resources

- ❖ Spring defines the ***Resource*** interface which offers useful methods for loading URL resources (*getURL()*, *exists()*, *isOpen()*, *createRelative()*, ...)
- ❖ Several useful implementations are also provided:
 - *ClassPathResource*: Loaded from a classpath
 - *ServletContextResource*: Loaded from a path relative to the root of a web application



ResourceLoader

❖ *ApplicationContext* implements the *ResourceLoader* interface which determines the resource loading strategy

❖ For example

```
Resource template = ctx.getResource("some/resource/path/myTemplate.txt");
```

- returns a *ClassPathResource* for a *ClassPathXmlApplicationContext*
- returns a *ServletContextResource* for a *WebApplicationContext*



Validation, Data Binding and Type Conversion

All Spring projects have support for :

- **Validation** of domain model classes.
Via *javax.validation*
- **Data Binding** allow to easily associate user entries with model classes
- **Type conversion** : String, number, Date, etc.



SpEL

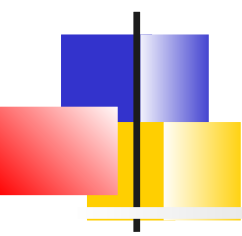
Spring Expression Language (SpEL) is a syntax for referencing beans defined in the container.

It allows access to bean's properties but also invocation of methods

```
@lombok.Data
public class FieldValueTestBean {

    // Injection de valeur
    @Value("#{ systemProperties['user.region'] }")
    private String defaultLocale;

}
```



Configuration via annotations

Configuration classes

@Component and stereotypes

Dependency injection

Configuration properties

Environment and profiles



Comparison with XML

- ❖ Instead of XML, it is possible to use annotations in the bean class.
- ❖ Each approach has its pros and cons
 - Annotations take advantage of their placement context which makes configuration more concise
 - XML allows wiring without requiring source code recompilation
 - Annotated classes are no longer simple POJOs
 - With annotations, the configuration is decentralized and becomes more difficult to control

Annotations are processed before XML configuration so XML configuration can override configuration by annotations



Configuration classes

Beans may be defined in Java classes annotated by **@Configuration**

These class contains methods annotated with **@Bean** which are used to instantiate Beans

@Configuration

```
public class AppConfig {  
    @Bean  
    public MyService myService() {  
        return new MyServiceImpl();  
    }  
}
```




Composition of configuration

@Import annotation allows to include other *@Configuration* classes

```
@Configuration
```

```
public class ConfigA {  
    public @Bean A a() { return new A(); }  
}
```

```
@Configuration
```

```
@Import(ConfigA.class)
```

```
public class ConfigB {  
    public @Bean B b() { return new B(); }  
}
```

```
-
```

```
ApplicationContext ctx = new  
    AnnotationConfigApplicationContext(ConfigB.class);
```



Boot

2 alternatives for the framework to process annotations:

- Indicate the location of the configuration classes
- Indicate a package to scan

In practice (with SpringBoot), it will be the 2nd alternative that will be used.



Examples

Indication of a *@Configuration* class

```
public static void main(String[] args) {  
    ApplicationContext ctx = new  
    AnnotationConfigApplicationContext(AppConfig.class);  
    MyService myService = ctx.getBean(MyService.class);  
    myService.doStuff();  
}
```

Scan of package(s) :

```
public static void main(String[] args) {  
    AnnotationConfigApplicationContext ctx = new  
    AnnotationConfigApplicationContext();  
    ctx.scan("com.acme");  
    ctx.refresh();  
    MyService myService = ctx.getBean(MyService.class);  
}
```



Attributes of *@Bean*

@Bean define 3 attributes :

name : aliases of bean

init-method : Method called after bean initialization by Spring

destroy-method : Method called before the bean is destroyed by Spring

@Configuration

```
public class AppConfig {  
    @Bean(name={"foo", "super-foo"}, initMethod = "init")  
    public Foo foo() {  
        return new Foo();  
    }  
    @Bean(destroyMethod = "cleanup")  
    public Bar bar() {  
        return new Bar();  
    }  
}
```



Dependency injection

In this case, dependency injection is done simply by method calls

```
@Configuration
public class AppConfig {
    @Bean
    public Foo foo() {
        return new Foo(bar());
    }
    @Bean
    public Bar bar() {
        return new Bar();
    }
}
```



@Enable annotations

@Configuration classes are generally used to configure external resources such as a database, a connection to another system, ...

To make it easier to configure these resources, Spring provides ***@Enable*** annotations that configure the resource's default values.

Thus, configuration classes only have to customize the default configuration

It is a firstfruit of SpringBoot



Examples of *@Enable*

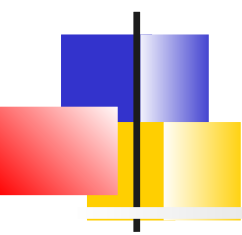
@EnableWebMvc : Default configuration for Spring MVC

@EnableCaching : Default configuration for cache features

@EnableScheduling : Default configuration for scheduling code execution

@EnableJpaRepositories : Scan of Repository classes used in Spring Data

...



Configuration via annotations

Configuration classes

@Component and stereotypes

Dependency injection

Configuration properties

Environment and profiles



Introduction

Other annotations are generally used to declare applicative beans

The main annotation for defining an application bean is ***@Component***, placed on the class



Example

@Component

```
public class SimpleMovieLister {  
    private MovieFinder movieFinder;  
    @Autowired  
    public SimpleMovieLister(MovieFinder movieFinder) {  
        this.movieFinder = movieFinder;  
    }  
}
```

@Component

```
public class JpaMovieFinder implements MovieFinder {  
    ...  
}
```



@ComponentScan

Spring can automatically detect classes corresponding to beans

Just add the **@ComponentScan** annotation and indicate a package to start the scan.

This is normally done on a *@Configuration* class

```
@Configuration
@ComponentScan(basePackages = "org.example")
public class AppConfig {
    ...
}
```



Stereotypes

Spring introduces other stereotypes:

- *@Component* is a generic stereotype for all components managed by Spring.
- ***@Repository***, ***@Service***, ***@Controller*** and ***@RestController*** are specializations of *@Component* for more specific use cases (persistence, service, and presentation layer)

Stereotypes are used to classify beans and optionally add generic behaviors to them.

Eg: Transactional behavior to all *@Service* components



@Scope

The **@Scope** annotation allows to specify one of Spring's predefined scopes or a custom scope

```
@Scope("prototype")
```

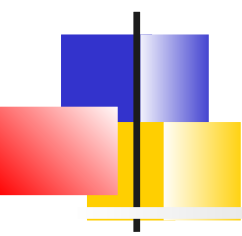
```
@Repository
```

```
public class MovieFinderImpl implements
```

```
    MovieFinder {
```

```
// ...
```

```
}
```



Configuration via annotations

Configuration classes
@Component and stereotypes
Dependency injection
Configuration properties
Environment and profiles



@Autowired

The **@Autowired** annotation is placed on setter methods, arbitrary methods, constructors

It asks Spring to inject a bean of the type of the argument

Usually, only one bean is a candidate for injection

@Autowired has an additional attribute : **required** (true by default)



Examples

```
public class SimpleMovieLister {
    private MovieFinder movieFinder;
    @Autowired
    public void setMovieFinder(MovieFinder movieFinder) { this.movieFinder = movieFinder;
}
// ...
}
...
public class MovieRecommender {
    private MovieCatalog movieCatalog;
    private CustomerPreferenceDao customerPreferenceDao;
    @Autowired
    public void prepare(MovieCatalog movieCatalog, CustomerPreferenceDao customerPreferenceDao)
    {
        this.movieCatalog = movieCatalog;
        this.customerPreferenceDao = customerPreferenceDao;
    }
    // ...
}
```




Examples

```
public class MovieRecommender {  
    @Autowired  
    private MovieCatalog movieCatalog;  
    private CustomerPreferenceDao customerPreferenceDao;  
    @Autowired  
    public MovieRecommender(CustomerPreferenceDao customerPreferenceDao) {  
        this.customerPreferenceDao = customerPreferenceDao;  
    }  
    // ...  
}  
...  
public class MovieRecommender {  
    @Autowired  
    private MovieCatalog[] movieCatalogs;  
    // ...  
}
```



@Qualifier

@Qualifier allows to select a candidate for auto-wiring among several possible ones

- The annotation takes as an attribute a String whose value must correspond to a configuration element of a bean



Example

```
public class MovieRecommender {  
    @Autowired  
    @Qualifier("main")  
    private MovieCatalog movieCatalog;  
    // ...  
}  
---  
<beans>  
    <context:annotation-config/>  
    <bean class="example.SimpleMovieCatalog">  
        <qualifier value="main"/>  
        <!-- .... --></bean>  
    <bean class="example.SimpleMovieCatalog">  
        <qualifier value="action"/>  
        <!-- .... --></bean>  
    <bean id="movieRecommender" class="example.MovieRecommender"/>
```



@Resource

@Resource allows to inject a bean by its name.

- The annotation takes the **name** attribute which must indicate the name of the bean
- If the name attribute is not specified, the name of the bean to inject corresponds to the name of the property



Examples

```
public class MovieRecommender {  
  
    @Resource(name="myPreferenceDao")  
    private CustomerPreferenceDao cpDao;  
  
    @Resource  
    private ApplicationContext context;  
  
    public MovieRecommender() {  
    }  
    // ...  
}
```



Callback methods

Spring supports **@PostConstruct** and **@PreDestroy** callbacks

```
public class CachingMovieLister {  
    @PostConstruct  
    public void populateMovieCache() {  
        // Initialization after construction...  
    }  
    @PreDestroy  
    public void clearMovieCache() {  
        // Clean before destruction...  
    }  
}
```



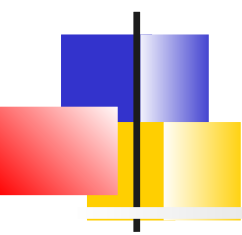
Annotations JSR 330 equivalence

Since version 3.0, Spring supports JSR 330 annotations.

@javax.inject.Inject is equivalent to *@Autowired*

@javax.inject.Named is equivalent to
@Component

@javax.inject.Singleton is equivalent to
@Scope("singleton")



Configuration via annotations

Configuration classes
@Component and stereotypes
Dependency injection
Configuration properties
Environment and profiles



Introduction

Spring can also inject simple values (basic types like String, Integer, ..) into bean's attributes :

- **@PropertySource** allows to indicate a *.properties* file to load configuration values
- **@Value** allows bean's attribute to be initialized with an SpEl expression

This requires the presence of a ***PropertySourcesPlaceholderConfigurer*** bean



Example

@Configuration

@PropertySource("classpath:/com/myco/app.properties")

public class AppConfig {

@Value("\${my.property:0}")

Integer myIntProperty ;

@Autowired

Environment env;

@Bean

public static **PropertySourcesPlaceholderConfigurer** properties() {
 return new PropertySourcesPlaceholderConfigurer();

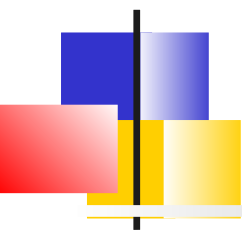
}

@Bean

public TestBean testBean() {
 TestBean testBean = new TestBean();
 TestBean.setIntProperty(myIntProperty) ;
 testBean.setName(env.getProperty("testbean.name"));
 return testBean;

}

}



Configuration via annotations

Configuration classes
@Component and stereotypes
Dependency injection
Configuration properties
Environment and profiles



Environment

The ***Environment*** interface is an abstraction modeling 2 aspects:

- ***Properties***: These are bean configuration properties.
They come from *.properties* , *.yml* files, command line argument or other...
- ***Profiles***: Named group of beans, beans are registered only if profile is enabled



Example

@Configuration

@Profile("development")

```
public class StandaloneDataConfig {

    @Bean
    public DataSource dataSource() {
        return new EmbeddedDatabaseBuilder()
            .setType(EmbeddedDatabaseType.HSQL)
            .addScript("classpath:com/bank/config/sql/schema.sql")
            .addScript("classpath:com/bank/config/sql/test-data.sql")
            .build();
    }
}
```

@Configuration

@Profile("production")

```
public class JndiDataConfig {

    @Bean(destroyMethod="")
    public DataSource dataSource() throws Exception {
        Context ctx = new InitialContext();
        return (DataSource) ctx.lookup("java:comp/env/jdbc/datasource");
    }
}
```



Activation of a profile

Programmatically:

```
AnnotationConfigApplicationContext ctx = new
    AnnotationConfigApplicationContext();
ctx.getEnvironment().setActiveProfiles("development");
ctx.register(SomeConfig.class, StandaloneDataConfig.class,
    IndiDataConfig.class);
ctx.refresh()
```

Java system property

```
-Dspring.profiles.active="profile1,profile2"
```

With Spring Boot :

```
--spring.profiles.active=profile1,profile2
```



Spring AOP

AOP Concepts

Features of Spring AOP

AOP via XML

AOP via *@AspectJ*

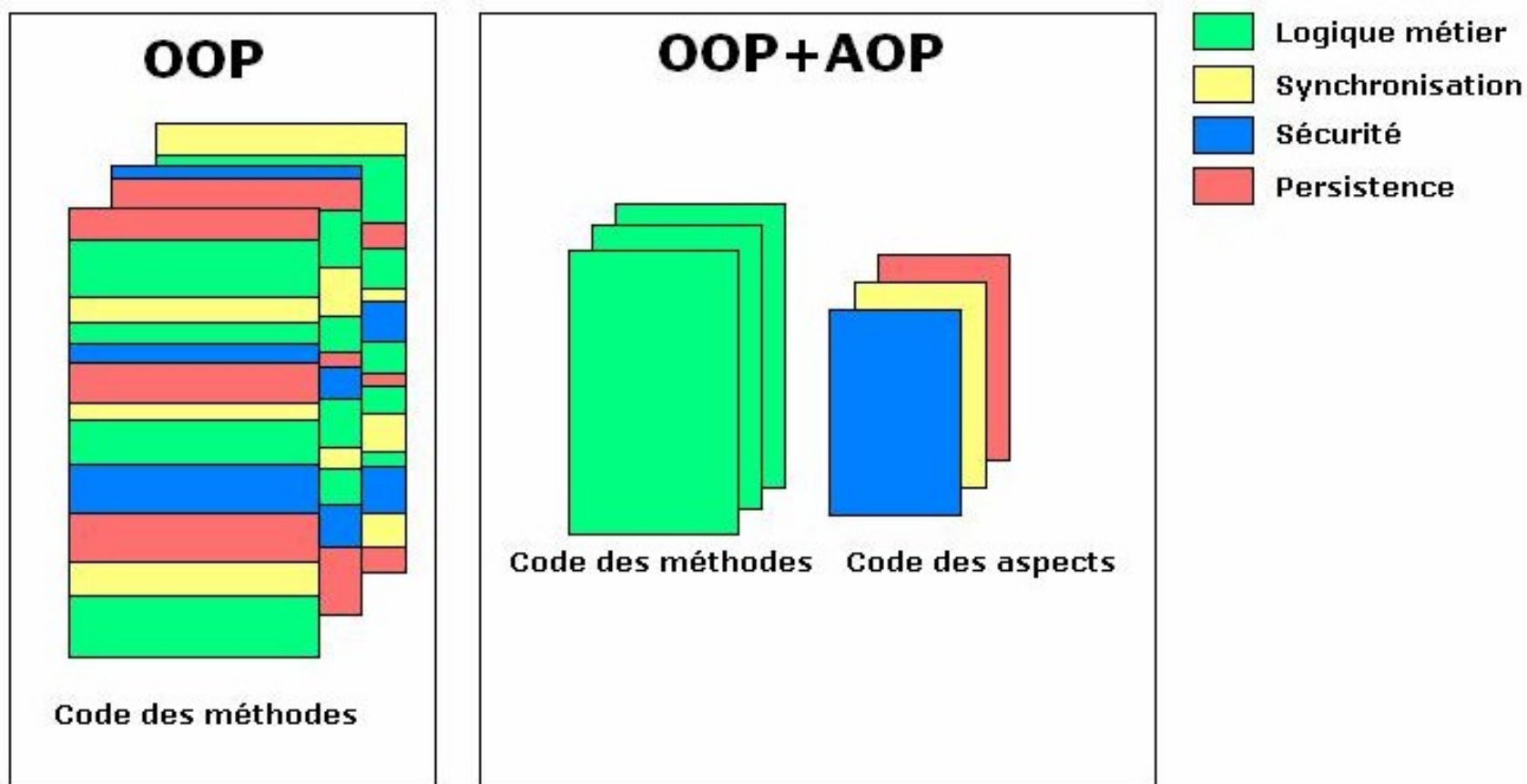


Introduction

- ❖ AOP (Aspect Oriented Programming) allows to circumvent certain limitations of object-oriented programming (OOP) :
 - With OOP many lines of code are dedicated to **crosscutting concerns** (like logging, exception handling, check input parameters, , etc.
- ❖ These lines of code are scattered in the methods of classes and that do not favor
 - Neither reuse
 - Neither evolutivity .
- ❖ AOP allows to centralize the code concerning the same crosscutting concern or *aspect*



Cross-cutting concerns





AOP benefits

- ❖ Aspect Oriented Programming (AOP) provides a very flexible solution for implementing cross-cutting technical services to application code such as transaction management, tracing, profiling and security.



Terminology

- ❖ An **aspect** is a factorization of the code applying to several classes (Example: transaction management).
- ❖ A **join point**: This is a specific place in the execution flow, where it is valid to insert an advice.
For example, it is not possible to insert an advice in the middle of the code of a function.
On the other hand, we can do it before, around, instead of or after the call to the function
- ❖ An **advice** : Some code that will be executed at certain junction points.
- ❖ An **interceptor**: This is a special type of advice that runs before and after a method. Interceptors are organized into an intercept chain.
- ❖ A **pointcut**: This is the place in the software where a plugin is inserted by the aspect weaver.
A pointcut is an expression that it is resolved in several join points.



Terminology (2)

- ❖ **Introduction** : Also called "inter-type declaration", a way to add additional methods to an object. Spring allows to introduce interfaces
- ❖ **Target object**: The object on which we placed plugins. The implementation is based on the proxy pattern, so any target object is associated with a proxy.
- ❖ **AOP proxy**: The object created by the AOP framework and which implements the methods of the advice.
- ❖ **Weaving**: This is the processing that binds aspects to objects. This process can be run at compile time or dynamically at runtime. Spring performs weaving at runtime.



Use cases

- ❖ The most classic application of AOP concerns the **declarative management of transactions**. Advices are of type around and intercept method calls to:
 - On the way: Create a transaction or attach to a current transaction
 - On return: Commit or rollback the transaction
- ❖ Another use case can be applying specific **security controls**.
- ❖ You can also insert advices for **debugging or profiling** applications during the development phases
- ❖ Advices that handle exceptions, Ex : interceptors that send emails on fatal errors,



Spring AOP

AOP Concepts

Features of Spring AOP

AOP via XML

AOP via *@AspectJ*



Introduction

- ❖ Spring AOP is implemented entirely in Java and no prior compilation phase is necessary.
- ❖ It does not rely on a hierarchical class loader and can therefore be used in a Tomcat or JavaEE server
- ❖ The main purpose of Spring AOP is
 - Provide cross-functional services to simple POJOs.
Ex : transaction service
 - To allow the implementation of custom aspects
- ❖ 2 alternatives to use Spring AOP
 - Declare aspects via XML
 - Using annotations from *@AspectJ*



Spring AOP features

❖ Spring AOP supports

- **Method interception**: Specific behaviors can then be added before and after invocations of application methods.
- **Introduction**: Specifying an advice result in the implementation of an additional interface.
- **Static and dynamic pointcuts**: Pointcuts expression can be used by several advices.
 - Static pointcuts are related to method signatures.
 - Dynamic pointcuts can take into account the arguments of the methods.



Types of advices

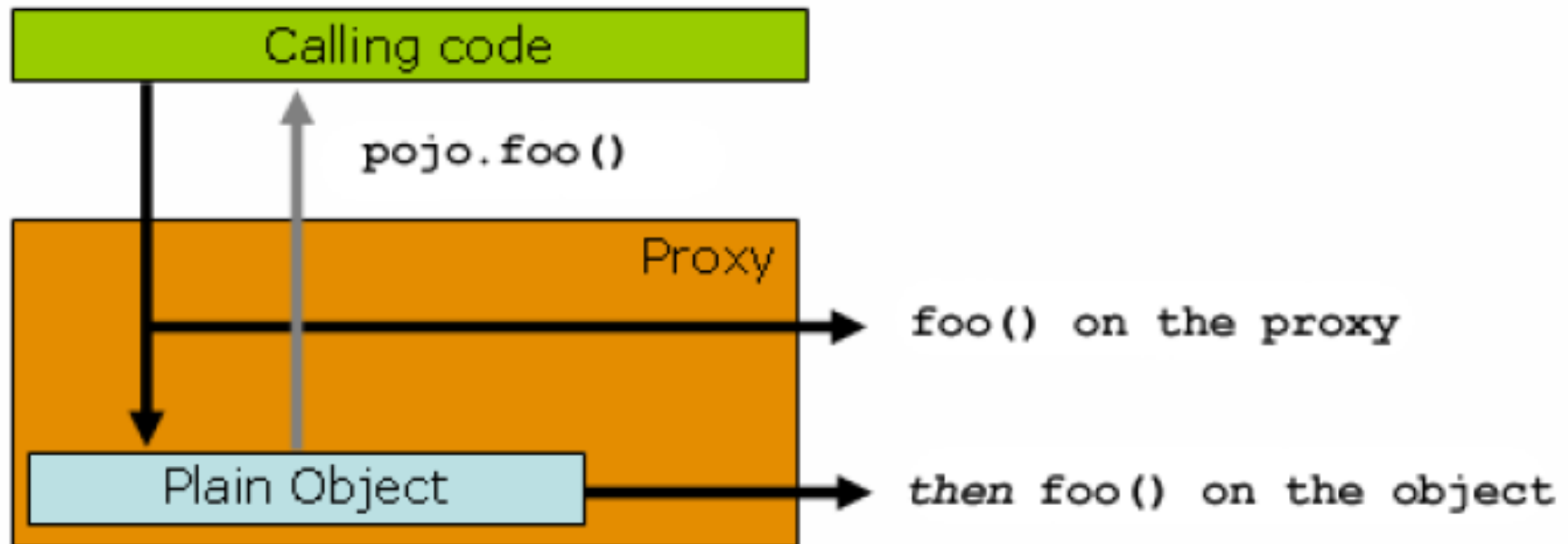
- ❖ Spring injects advices on object instances and not at the class level. It is thus possible to have different instances of the same class on which the same plugins are not applied.
- ❖ 5 types of advices are available :
 - **Before advice** : Execution before the pointcut. The pointcut method is always executed unless the advice throws an exception
 - **After returning advice** : Execution after the pointcut method executes normally.
 - **After throwing advice** : Execute after the pointcut method if it throws an exception
 - **After (finally) advice** : Execution after the pointcut method returns anyway.
 - **Around advice** : Execution before and after the pointcut; it is the most generic advice. It can also decide not to run the pointcut if the advice throws an exception.



Implementation

- ❖ Spring's implementation of AOP is based on Proxies:
 - Either by using the JDK's dynamic proxies (*java.lang.reflect.Proxy*). This method is used when an interface exists.
 - Or by generating proxy classes at runtime using the CGLIB library. If the aspects are applied on concrete classes

Proxy mechanism





Direct method calls

- ❖ Beware of direct method calls. In the following example the plugins assigned to the *bar()* method are not executed when the *foo()* method is called

```
public class SimplePojo implements Pojo {  
    public void foo() {  
        this.bar();  
    }  
    public void bar() {  
        // some logic...  
    }  
}
```



Spring AOP

AOP Concepts
Features of Spring AOP
AOP via XML
AOP via *@AspectJ*



Namespace

❖ To use the *aop* namespace, the ***spring-aop*** must be imported :

```
<?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:aop="http://www.springframework.org/schema/aop"
xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd">
```



<aop: elements

- ❖ All AOP-related configurations are included in one or more **<aop:config>** elements which contains the sequenc of following sub-elements :
 - **<aop:pointcut/>**: Used to express pointcuts that can be used by different aspects.
 - **<aop:advisor/>**: Specific to Spring, this is a language simplification to express an aspect containing a single advice.
 - **<aop:aspect/>**: Allows you to specify aspects, it also contains sub-elements **<aop:pointcut/>**.



Aspect declaration

- ❖ An aspect is a simple Spring bean to which is added the definition of pointcuts and advices.
- ❖ An aspect is therefore declared thanks to the `<aop:aspect>` element which indicates the reference to a Spring bean

```
<aop:config>
  <aop:aspect id="myAspect" ref="aBean">
    ...
  </aop:aspect>
</aop:config>
<bean id="aBean" class="...">
  ...
</bean>
```




Pointcuts

❖ The *AspectJ* syntax is used:

```
<aop:config>
```

```
<aop:pointcut id="businessService"
```

```
expression="execution(*com.xyz.myapp.service.*(..))"/>
```

```
</aop:config>
```



Advices

❖ Advices elements availables are :

- *<aop:before/>*
- *<aop:after-returning/>*
- *<aop:after-throwing/>*
- *<aop:after/>*
- *<aop:around/>*



Examples

```
<aop:aspect id="afterThrowingExample" ref="aBean">
  <aop:before pointcut-ref="dataAccessOperation"
    method="doAccessCheck"/>
  <aop:after-returning pointcut-ref="dataAccessOperation"
    returning="retVal" method="doAccessCheck"/>
  <aop:after-throwing pointcut-ref="dataAccessOperation"
    throwing="dataAccessEx" method="doRecoveryActions"/>
  <aop:after pointcut-ref="dataAccessOperation"
    method="doReleaseLock"/>
  <aop:around pointcut-ref="businessService"
    method="doBasicProfiling"/>

</aop:aspect>
```



Example

- ❖ Performance monitoring on business methods:

```
<bean id="performanceMonitor"
      class="com.example.PerformanceMonitor"/>
<aop:config>
  <aop:aspect ref="performanceMonitor">
    <aop:around pointcut="execution(public *
        com.example.Service+.*(..))" method="monitor" />
  </aop:aspect>
</aop:config>
```

- ❖ The *monitor()* method of *com.example.PerformanceMonitor* is called as soon as a method of the *com.example.Service* class or its subclasses is invoked.



Example (2)

❖ The *monitor()* method

- starts a timer,
- authorizes the normal execution of the target method (via the *proceed()* method),
- then stops the timer when the target method returns,
- saves the measurement then returns the target method value:

```
public Object monitor(ProceedingJoinPoint pjp) throws Throwable {  
    long start = System.nanoTime();  
    try {  
        return pjp.proceed();  
    } finally {  
        long time = System.nanoTime() - start;  
        // do something with time...  
    }  
}
```



Spring AOP

AOP Concepts
Features of Spring AOP
AOP via XML
AOP via *@AspectJ*



Enabling *@AspectJ*

- ❖ Ensure the presence of ***aspectjweaver.jar***
org.aspectj :aspectjweaver
- ❖ Then allow support of *@AspectJ* :
 - via XML

<aop:aspectj-autoproxy/>

- via Java

@Configuration

@EnableAspectJAutoProxy

public class AppConfig



Usage of *@AspectJ*

- ❖ Once *@AspectJ* is enabled, you must:
 - Define an aspect, via ***@Aspect***
 - If necessary, declare the pointcuts
 - Declare advices with annotations



Aspect declaration

```
package org.xyz;  
import org.aspectj.lang.annotation.Aspect;  
@Aspect  
public class NotVeryUsefulAspect {  
}
```

- ❖ Aspects contain methods and attributes like any Java class plus they declare pointcuts, advices and introductions



Pointcut declaration

❖ The declaration of a pointcut consists of two parts:

- **Its signature**: method name and parameters.
The return type is always void
- **An expression**: Allows to determine the methods on which the aspect will be applied

```
@Pointcut("execution(* transfer(..))")// the pointcut expression
```

```
private void anyOldTransfer() {}// the pointcut signature
```



Pointcut expression

- Spring AOP only supports a subset of the AspectJ syntax. wildcard (*) is supported:

execution : Some method signatures

within : Some types

this : A given type of proxy

target : A given type of the target object

args : A given type of method arguments

@target : A given annotation of the target object

@args : A given annotation of the method arguments

@within : A given annotation of a type of the target object

@annotation : A given annotation of methods

bean : A Spring bean name (Spring specific)



Examples

execution(public * *(..)) : All public methods

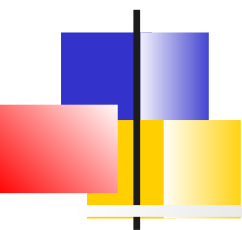
execution(* com.xyz.service.*.*(..)) : All methods
define in the service package

within(com.xyz.service..*) : All methods define in the
service package and its sub-packages

this(com.xyz.service.AccountService) : All the methods
of beans if its proxy implements an interface

target(com.xyz.service.AccountService) : All the
methods of beans implementing the AccountService
interface

args(java.io.Serializable) : All methods wich takes a
single *Serializable* argument



Example *SystemArchitecture*

```
@Aspect
public class SystemArchitecture {
    @Pointcut("within(com.xyz.someapp.web..*)")
    public void inWebLayer() {}

    @Pointcut("within(com.xyz.someapp.service..*)")
    public void inServiceLayer() {}

    @Pointcut("within(com.xyz.someapp.dao..*)")
    public void inDataAccessLayer() {}

    @Pointcut("execution(* com.xyz.someapp.service.*.*(..))")
    public void businessService() {}

    @Pointcut("execution(* com.xyz.someapp.dao.*.*(..))")
    public void dataAccessOperation() {}
}
```



Advices declaration

- ❖ An advice is associated to a pointcut and is executed before and/or after
- ❖ And advice can be associated
 - To the reference of the pointcut

```
@Before("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")  
public void doAccessCheck() { ... }
```

- Directly to an pointcut expression

```
@Before("execution(* com.xyz.myapp.dao.*.*(..))")  
public void doAccessCheck() { ... }
```



Advice Annotations

❖ Available annotations are :

- ***@Before***
- ***@AfterReturning***
- ***@AfterThrowing***
- ***@After (finally)***
- ***@Around***



Features of after advice

- ❖ *@AfterReturning* can use the value returned by the method:

```
@AfterReturning(  
pointcut="com.xyz.myapp.SystemArchitecture.dataAccessOperation()",  
returning="retVal")  
public void doAccessCheck(Object retVal) {  
    // ...  
}
```

- ❖ It is possible to restrict the execution of *@AfterThrowing* to a certain type of exception:

```
@AfterThrowing(  
pointcut="com.xyz.myapp.SystemArchitecture.dataAccessOperation()",  
throwing="ex")  
public void doRecoveryActions(DataAccessException ex) {  
    // ...  
}
```




Annotation *@Around*

- ❖ The first argument of an *Around* method must be of type *ProceedingJoinPoint*
 - The call to the method *proceed()* on this argument starts the execution of the method of the target object
 - You can pass as an argument an array of objects that will serve as arguments to the method
 - The value returned by the *@Around* method will be the value received by the caller



SpringBoot

Auto-configuration

Starters SpringBoot

Project structure and main annotations

Configuration properties



Introduction

Spring Boot was designed to make it easy to start and develop new Spring applications.

- does not require any XML configuration
- From the first line of code, we have a functional application

=> It offers a development experience that simplifies to the extreme the use of existing technologies



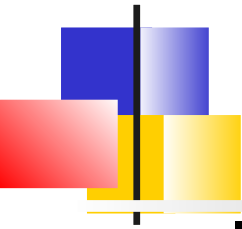
Auto-configuration

The main concept of SpringBoot is **auto-configuration**

- SpringBoot is able to automatically detect the nature of the application and configure the necessary Spring beans

This allows you to start quickly with a default configuration and gradually override the default configuration for application needs

The mechanisms are different depending on the language: Groovy, Java or Kotlin



Java

Dependencies management

In a Java environment, Spring Boot simplifies the management of dependencies and their versions:

- It organizes functionalities into modules. Each module come with its managed dependencies

A wizard named "***Spring Initializr***", is used to select the modules and generate Maven or Gradle configurations

=> For developers, this mechanism simplifies to the extreme the management of dependencies and their versions

Only one version left to manage: SpringBoot

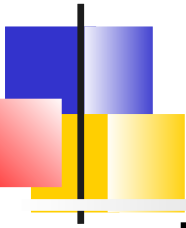


Auto-configuration (Java)

Depending on the libraries present at runtime, Spring Boot creates all the necessary technical beans with a default configuration. For examples

- If Spring notices that web libraries are present, it starts an embedded Tomcat server on port 8080 and configures all the technical beans required to develop a Web application or Rest API
- If Spring notices that the Postgres driver is in the classpath, it tries to automatically creates a pool of connections to the database
- etc.

=> The project is therefore executable with the minimum of prior configuration



Configuration Customization

Default configuration can be overridden by different means

- **Configuration properties** that modify the default values of technical beans via:
 - Environment variables
 - Command line arguments
 - External configuration files (.properties or .yml).
Different files can be activated according to profiles
- Specific classes of the framework (generally named Configurer or Customizer)



Auto-configuration

Spring Boot's auto-configuration mechanism is implemented based on:

- Classic *@Configuration* class that define beans
- And *@Conditional* annotations that specify the conditions for the configuration to activate

So when Spring Boot starts, the conditions are evaluated and if they are met, the corresponding integration beans are instantiated and configured.



Conditional annotations

Conditions can be based on:

- The presence or absence of a class:
@ConditionalOnClass and **@ConditionalOnMissingClass**
- The presence or absence of a bean:
@ConditionalOnBean or **@ConditionalOnMissingBean**
- A property :
@ConditionalOnProperty
- The presence of a resource:
@ConditionalOnResource
- Whether the app is a web app or not:
@ConditionalOnWebApplication or
@ConditionalOnNotWebApplication
- An **SpEL** expression



Example Apache SolR

```
@Configuration
@ConditionalOnClass({ HttpSolrClient.class, CloudSolrClient.class })
@EnableConfigurationProperties(SolrProperties.class)
public class SolrAutoConfiguration {

    private final SolrProperties properties;

    private SolrClient solrClient;

    public SolrAutoConfiguration(SolrProperties properties) {
        this.properties = properties;
    }

    @Bean
    @ConditionalOnMissingBean
    public SolrClient solrClient() {
        this.solrClient = createSolrClient();
        return this.solrClient;
    }

    private SolrClient createSolrClient() {
        if (StringUtils.hasText(this.properties.getZkHost())) {
            return new CloudSolrClient(this.properties.getZkHost());
        }
        return new HttpSolrClient(this.properties.getHost());
    }
}
```



Consequences

The Solar starter pulls

- Conditional Configuration Classes
- Sonar libraries

SolR integration beans are created and can be injected into application code.

```
@Component
public class MyBean {

    private SolrClient solrClient;

    public MyBean(SolrClient solrClient) {
        this.solrClient = solrClient;
    }
}
```



SpringBoot

Auto-configuration

Starters SpringBoot

Project structure and main annotations

Configuration properties



Starters

Developers use starter-modules which provide:

- A set of libraries dedicated to a type of application.
- Technical beans that provide configurable integration services

Spring provides starter-modules for any type of problem

A wizard lists all available starters:

<https://start.spring.io/>



Most used starters

Web

- *-**web** : Web application or API REST
- *-**reactive-web** : Reactive web application web or Reactive API REST
- *-**web-services** : Web services SOAP

Core :

- *-**logging** : logback (default)
- *-**test** : Junit, Mockito, ... (default)
- *-**devtools** : Dev environment
- *-**lombok** : Lombok library
- *-**configuration-processor** : Applicative properties management



Security

*-**security** : Spring Security, security apply to web and business layer

*-**oauth2-client** : To obtain an *oAuth* token from an authorization server

*-**oauth2-resource-server** : Secure URLs via *oAuth2*

*-**ldap** : LDAP integration

*-**okta** : Okta integration



Persistence

Access to persistent store with Spring Data

*-**jdbc** : Simple mapping with JDBC

*-**jpa** : Hibernate and JPA

*-**<drivers>** : Most used JDBC drivers (MySQL, Postgres, H2, HyperSonic, ..)

*-**data-cassandra**, *-**data-reactive-cassandra**: Cassandra

*-**data-neo4j** : Neo4j

*-**data-couchbase** *-**data-reactive-couchbase** : CouchBase

*-**data-redis** *-**data-reactive-redis** : Redis

*-**data-geode** : Geode

*-**data-elasticsearch** : ElasticSearch

*-**data-solr**: SolR

*-**data-mongodb** *-**data-reactive-mongodb** : MongoDB

*-**data-rest** : Restful APIs over Spring Data



Messaging

- *-**integration**: Spring Integration
- *-**kafka**: Apache Kafka
- *-**kafka-stream**: Kafka Stream
- *-**amqp**: Spring AMQP et Rabbit MQ
- *-**activemq** : Apache ActiveMQ
- *-**artemis** : Apache Artemis
- *-**websocket** : Intégration with STOMP and SockJS
- *-**camel** : Apache Camel



UI Web

UI Web, Mobile

- *-**thymeleaf** : MVC with *Thymeleaf*
- *-**mobile** : Spring Mobile
- *-**hateoas** : Application RESTFul with Spring Hateoas
- *-**jersey** : API Restful with JAX-RS and Jersey
- *-**websocket**: Spring WebSocket
- *-**mustache** : Spring MVC with Mustache
- *-**groovy-templates** : MVC with Groovy
- *-**freemarker**: MVC with freemarker



Other Starters

I/O

- *-**batch** : Batch framework
- *-**mail** : Sending and receiving mails
- *-**cache**: Support for cache
- *-**quartz** : Support for Scheduling

Ops

- *-**actuator** : Monitoring endpoints (REST or JMX)
- *-**spring-boot-admin** : UI over actuator



Spring Cloud

Cloud Services providers

Amazon, Google Cloud, Azure, Cloud Foundry, Alibaba

Micro-services architecture

Discovery, external configuration, load balancing, gateway, distributed monitoring and tracing, circuit breaker pattern, etc ...



SpringBoot

Auto-configuration

Starters SpringBoot

**Project structure and main
annotations**

Configuration properties



Project structure

No obligation but recommendations:

- Place the Main class in the root package,
All other packages must be sub-packages in order
to Spring to be able to scan the annotations
- Annotate it with:
 - @EnableAutoConfiguration
 - @ComponentScan
 - @Configuration
- Or simply :
 - **@SpringBootApplication**



Packages structure

com

+ - example

+ - myproject

+ - Application.java

+ - OneConfig.java

|

+ - domain

| + - Customer.java

| + - CustomerRepository.java

|

+ - service

| + - CustomerService.java

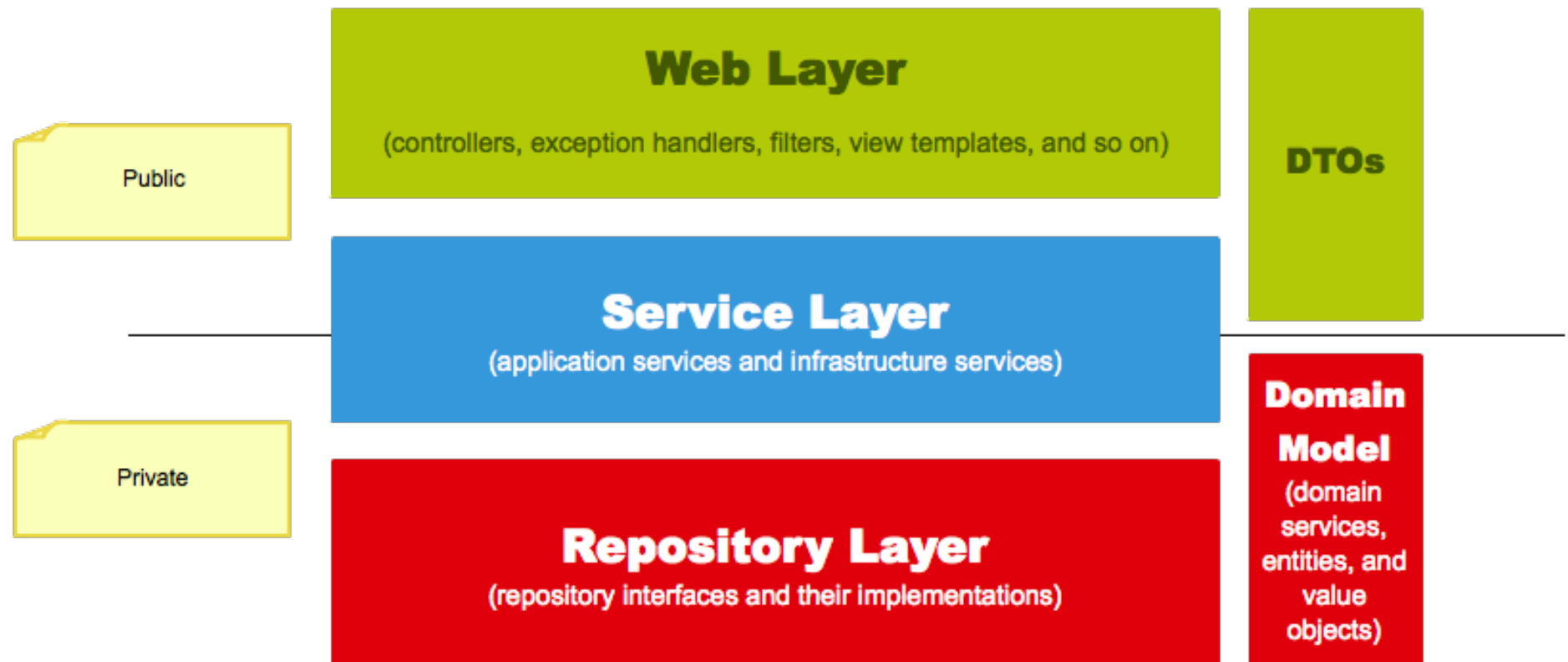
|

+ - web

+ - CustomerController.java



Classical layered web project





@Configuration

@Configuration tells Spring that the class can define Spring beans

- The Main class can be a good place for configuration
- But this one can be dispersed in several other classes
- The *@Import* annotation can be used to import the other configuration classes



Example

```
@Import(DataSourceConfig.class)  
@Configuration  
public class SimpleConfiguration {  
    @Autowired  
    Connection connection;  
  
    @Bean  
    Database getDatabaseConnection(){  
        return connection.getDBConnection();  
    }  
    // Mode code here....  
}
```



Auto-configuration

@EnableAutoConfiguration

automatically configures Spring beans based on dependencies that have been specified.

- Ability to disable auto-configuration for certain parts of the application.

Ex :

```
@EnableAutoConfiguration(exclude={DataSource  
    AutoConfiguration.class})
```



Beans and Dependency injection

It is possible to use the different techniques of Spring to define the beans and their dependency injections.

The simplest technique is usually a combination of:

- **@ComponentScan** annotation for Spring to find beans (Includes in @SpringBootApplication)
- The **@Component**, **@Service**, **@Repository**, **@Controller** annotations which allow to define beans
- The **@Autowired** annotation in a bean's constructor or on a declaration
- The use of implicit injection, final attribute + constructor parameter



Example

```
package com.example.service;

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

@Service
public class DatabaseAccountService implements AccountService {

    private final RiskAssessor riskAssessor;

    // Implicit injection
    public DatabaseAccountService(RiskAssessor riskAssessor) {
        this.riskAssessor = riskAssessor;
    }

    // ...

}
```



SpringBoot

Auto-configuration
Starter SpringBoot
Project structure and main annotations
Configuration properties



Configuration properties

Spring Boot allows bean configuration to be externalized:

- Ex: Externalize the DB URL, the configuration of a client, ...

You can use *.properties* or YAML files, environment variables or command-line arguments to set the properties values

These values can then be injected into the beans:

- Directly via **@Value** annotation
- Or associate to a structured object via **@ConfigurationProperties** annotation



Precedence

1. *spring-boot-devtools.properties* if devtools is enabled (SpringBoot)
2. Test properties
3. **The command line. Eg: --server.port=9000**
4. REST, Servlet, JNDI, JVM environment
5. **OS environment variables**
6. Properties with random values
7. **Profile-specific properties**
8. **application.properties , yml**
9. @PropertySource annotation in config
10. Default properties specified by
SpringApplication.setDefaultProperties



application.properties (.yml)

Properties files (application.properties/.yml) are typically placed in the following locations:

- A config subdirectory
- The current directory
- A config package in the classpath
- At the root of the classpath

By respecting these standard locations, SpringBoot finds them on its own



Filtered values

Configuration files support filtered values.

```
app.name=MyApp
```

```
app.description=${app.name} is a Boot app.
```

And random values:

```
my.secret=${random.value}
```

```
my.number=${random.int}
```

```
my.bignumber=${random.long}
```

```
my.uuid=${random.uuid}
```



Injection with *@Value*

The first way to retrieve a configured value is to use the annotation ***@Value***.

```
@Value("${my.property}")  
private String myProperty ;
```

In this case, no check is made on the effective value of the property



Validating properties

It is possible to force the validation of configuration properties at container initialization.

- Use a class annotated by ***@ConfigurationProperties*** and ***@Validated***
- Set ***javax.validation.constraints*** on class attributes



Example

@Component

@ConfigurationProperties("app")

@Validated

public class MyAppProperties {

@Pattern(regex = "\\d{3}-\\d{3}-\\d{4}")

private String adminContactNumber;

@Min(1)

private int refreshRate;

.....

}



Specific profile properties with **.properties*

Profile-specific properties (e.g. integration, production) can be in files named:

application-{profile}.properties



Specific profile properties with *.yml*

With YAML, it is possible to define several profiles in the same document.

```
server:
  address: 192.168.1.100
...
spring:
  config:
    activate:
      on-profile:
        -prod
server:
  address: 192.168.1.120
```



Include profiles

The property *spring.profiles.include* allows to include profiles in another profile.

For example :

```
my.property: fromyamlfile
```

```
spring.profiles: prod
```

```
spring.profiles.include:
```

- proddb
- prodm



Activating Profiles

Profiles are activated via the ***spring.profiles.active*** property which can be set:

- In a configuration file
- With the command line :
--spring.profiles.active=dev,hsqldb
- Programmatically, via:
SpringApplication.setAdditionalProfiles(...)

Multiple profiles can be activated simultaneously



Persistence

Concepts of SpringData

SpringData JPA

SpringData MongoDB



Introduction

Spring Data's objective is to provide a **simple and consistent programming model** for data access regardless of the underlying technology

Spring Data is therefore the project that wrap many specialized sub-projects (jdbc, JPA, Mongo, Cassandra, Elastic, ...)



Benefits of *SpringData*

Spring Data offers :

- An abstraction of the notion of **repository** and object mapping
- **Dynamic query generation** based on method naming rules
- Low level utility classes : ****Template.java***
- A support for the **audit** (Creation date, last change)
- The ability to embed **repository-specific code**
- Java or XML configuration
- Automatic integration with Spring MVC controllers via *SpringDataRest*



Repository interfaces

Spring Data's central interface is ***Repository***
(It's a marker class)

The interface takes arguments of type

- the persistent class of the domain
- its id.

CrudRepository subinterface adds CRUD methods

Technology-specific abstractions are also available *JpaRepository*, *MongoRepository*, ...



CrudRepository methods

```
public interface CrudRepository<T, ID extends Serializable>
    extends Repository<T, ID> {

    <S extends T> S save(S entity);

    T findOne(ID primaryKey);

    Iterable<T> findAll();

    Long count();

    void delete(T entity);

    boolean exists(ID primaryKey);

    // ... more functionality omitted.
}
```



Determining the request

After extending the interface, it is possible to define methods to make specific queries

At runtime Spring provides a bean implementing the provided interface and methods.

Spring must deduce the queries to perform:

- Either from the ***name*** of the method
- Either from the ***@Query*** annotation



Example

```
public interface MemberRepository
    extends JpaRepository<Member, Long> {

    /**
     * All memberes ayant un email particulier.
     * @param email
     * @return
     */
    public List<Member> findByEmail(String email);

    /**
     * Chargement de la jointure one2Many.
     * @param id
     * @return
     */
    @Query("from Member m left join fetch m.documents where m.id =:id")
    public Member fullLoad(Long id);
```




Query methods

To be recognized by Spring methods must be prefixed by :

- *find*By** : select objects
- *count*By** : count objects
- *delete*By** : delete objects
- *get*By** : Retrieve proxies

The first *** may indicate a flag (like *Distinct*)

The term **By** marks the end of the identification of the type of query

The rest is parsed and specify the **where** and optionally the **orderBy** clauses



Parsing result

Method names usually consist of entity properties combined with AND and OR

Operators can also be specified: *Between*, *LessThan*, *GreaterThan*, *Like*

The *IgnoreCase* flag can be set individually to properties or globally

```
findByLastnameIgnoreCase(...)
```

```
findByLastnameAndFirstnameAllIgnoreCase(...)
```

The order clause of the query can be specified by adding *OrderBy(Asc/Desc)* at the end of the method



Properties expression

Properties can only refer to direct properties of entities

It is however possible to reference nested properties:

```
List<Person> findByAddressZipCode(ZipCode zipCode);
```

Or if ambiguity

```
List<Person> findByAddress_ZipCode(ZipCode  
    zipCode);
```



Parameters management

In addition to property parameters, SpringBoot is able to recognize **Pageable** or **Sort** type parameters to apply pagination and sorting dynamically

Return values can be:

- Page (knows the total number of elements by making a count query)
- Slice only knows if there next records

```
Page<User> findByLastname(String lastname, Pageable pageable);  
Slice<User> findByLastname(String lastname, Pageable pageable);  
List<User> findByLastname(String lastname, Sort sort);  
List<User> findByLastname(String lastname, Pageable pageable);
```



Limit

The ***first*** and ***top*** keywords limit the returned entities

They can be specified with a numeric

```
User findFirstByOrderByLastnameAsc();  
Slice<User> findTop3ByLastname(String lastname,  
                                Pageable pageable);
```



Keywords supported for JPA

And, Or Is, Equals, Between,
LessThan, LessThanEqual,
GreaterThan, GreaterThanEqual,
After, Before, IsNull,
IsNotNull, NotNull, Like,
NotLike, StartingWith,
EndingWith, Containing, OrderBy,
Not, In, NotIn, True, False,
IgnoreCase



Use of JPA *NamedQuery*

With JPA the name of the method may match a *NamedQuery*.

```
@Entity
@NamedQuery(name = "User.findByEmailAddress",
    query = "select u from User u where u.emailAddress = ?1")
public class User {
}

public interface UserRepository extends JpaRepository<User, Long> {
    User findByEmailAddress(String emailAddress);
}
```



Use of *@Query*

The query can also be expressed in the query language of the repository via the **@Query** annotation:

- Highest priority method
- Has the advantage of being located on the Repository class

```
public interface UserRepository extends JpaRepository<User, Long> {  
  
    @Query("select u from User u where u.emailAddress = ?1")  
    User findByEmailAddress(String emailAddress);  
  
    @Query("select u from User u where u.firstname like %?1")  
    List<User> findByFirstnameEndsWith(String firstname);  
  
    @Query("select u.id, LENGTH(u.firstname) as fn_len from User u where u.lastname like ?1%")  
    List<Object[]> findByAsArrayAndSort(String lastname, Sort sort);  
  
    @Query("select u from User u where u.firstname = :firstname or u.lastname = :lastname")  
    User findByLastnameOrFirstname(@Param("lastname") String lastname,  
                                    @Param("firstname") String firstname);  
}
```




Persistence

Concepts of SpringData
SpringData JPA
SpringData MongoDB



Autoconfiguration Spring Boot

spring-boot-starter-data-jpa provide the following dependencies :

- Hibernate
- Spring Data JPA .
- Spring ORMs

By default, all classes annotated by JPA's annotations are scanned and taken into account

Scan starting location can be narrowed with
@EntityScan

Reminders: Entity Classes and Associations

@Entity

```
public class Theme {  
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long id;  
    private String label;  
    @OneToMany(cascade = CascadeType.ALL)  
    private Set<MotClef> motclefs = new HashSet<MotClef>();  
}
```

@Entity

```
public class MotClef {  
    @Id  
    private Long id;  
    private String mot;  
  
    public MotClef(){}  
}
```



Datasource configuration

To access a relational DB, Java uses the notion of ***DataSource*** (interface representing a pool of DB connections)

A data source is configured via:

- A JDBC URL
- A database account
- A JDBC driver
- Pool sizing parameters



Support for embeddeed databases

Spring Boot can automatically configure H2, HSQL and Derby databases.

You don't need to provide a connection URL, the Maven dependency is enough:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
  <groupId>org.hsqldb</groupId>
  <artifactId>hsqldb</artifactId>
  <scope>runtime</scope>
</dependency>
```



Production database

Production bases can also be easily configured.

The only required properties to provide are :

`spring.datasource.url=jdbc:mysql://localhost/test`

`spring.datasource.username=dbuser`

`spring.datasource.password=dbpass`

See *DataSourceProperties* to see all the availables properties

The underlying pool implementation is Hikari by default in Spring Boot 2.

This can be overridden by the property
spring.datasource.type



Pool configuration

Properties are also specific to the pool implementation used.

For example for Hikari:

```
# Timeout in ms to obtain a connection
spring.datasource.hikari.connection-
  timeout=10000
```

```
# Size of the pool
spring.datasource.hikari.maximum-pool-size=50
spring.datasource.hikari.minimum-idle= 10
```



Property *ddl-auto*

Embedded data base are fully created at startup and deleted when the application stops. (*ddl-auto=create-drop*)

- They can also be initialized with data by providing an *import.sql* file

For other databases, we may specify the property :

spring.jpa.hibernate.ddl-auto

- 5 possible values : *none, validate, update, create, create-drop*

Other native properties of Hibernate can be specified via the prefix *spring.jpa.properties.**

Ex :

```
spring.jpa.properties.hibernate.globally_quoted_identifiers=true
```

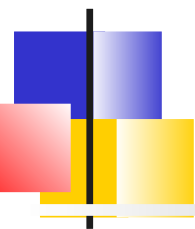



Transactional behavior

By default, repository methods are transactional.

For read operations, the *readOnly* flag is set.

All other methods are configured with a simple *@Transactional* so that the default transaction configuration applies



@Transactional and *@Service*

It is common to use a facade (@Service bean) to implement a business functionality requiring several calls to different Repositories

The ***@Transactional*** annotation allow to delimit a transaction for business operations.



Example

@Service

```
class UserManagementImpl implements UserManagement {  
  
    private final UserRepository userRepository;  
    private final RoleRepository roleRepository;  
  
    public UserManagementImpl(UserRepository userRepository,  
        RoleRepository roleRepository) {  
        this.userRepository = userRepository;  
        this.roleRepository = roleRepository;  
    }  
}
```

@Transactional

```
public void addRoleToAllUsers(String roleName) {  
  
    Role role = roleRepository.findByName(roleName);  
  
    for (User user : userRepository.findAll()) {  
        user.addRole(role);  
        userRepository.save(user);  
    }  
}
```



Templates

JdbcTemplate and
NamedParameterJdbcTemplate
beans are auto-configured and can
therefore be injected

Their behavior can be customized by
*spring.jdbc.template.** properties

Ex :

```
spring.jdbc.template.max-rows=500
```



Example

@Repository

```
public class UserDaoImpl implements UserDao {
```

```
    private final String INSERT_SQL = "INSERT INTO USERS(name, address, email) values(:name,:email)";
```

```
    private final String FETCH_SQL_BY_ID = "select * from users where record_id = :id";
```

@Autowired

```
private NamedParameterJdbcTemplate namedParameterJdbcTemplate;
```

```
public User create(final User user) {
```

```
    KeyHolder holder = new GeneratedKeyHolder();
```

```
    SqlParameterSource parameters = new MapSqlParameterSource()
```

```
        .addValue("name", user.getName())
```

```
        .addValue("email", user.getEmail());
```

```
    namedParameterJdbcTemplate.update(INSERT_SQL, parameters, holder);
```

```
    user.setId(holder.getKey().intValue());
```

```
    return user;
```

```
}
```

```
public User findUserById(int id) {
```

```
    Map parameters = new HashMap();
```

```
    parameters.put("id", id);
```

```
    return namedParameterJdbcTemplate.queryForObject(FETCH_SQL_BY_ID, parameters, new UserMapper());
```

```
}
```

```
}
```



JDBC or JPA layer

Spring Data does not prevent to use JDBC or JPA layers directly :

- With JDBC, just inject the *DataSource*
- With JPA, inject *EntityManager* or *EntityManagerFactory*



Persistence

Concepts of SpringData
SpringData JPA

SpringData MongoDB



Introduction

Spring Boot provides automatic configurations for *Redis*, *MongoDB*, *Neo4j*, *Elasticsearch*, *Solr* and *Cassandra*;

For example, *for MongoDB*

`spring-boot-starter-data-mongodb`



Connection to MongoDB

SpringBoot creates a ***MongoDbFactory*** bean connecting to the URL *mongodb://localhost/test*

The ***spring.data.mongodb.uri*** property allows to change the URL

- The other alternative is to declare your own *MongoDbFactory* or a *Mongo* type bean



Entity

Spring Data offers an ORM between MongoDB documents and Java objects.

A domain class can be annotated by **@Id**:

```
import org.springframework.data.annotation.Id;
public class Customer {
    @Id
    public String id;
    public String firstName;
    public String lastName;

    public Customer() {}
    ...
    // getters and setters
}
```



Mongo Repository

Spring Data also offers Repository implementations for NoSQL databases

Ex : *MongoRepository*

The programming model is then similar to JPA



Usage

```
@Controller
public class MyController {

    @Autowired
    private CustomerRepository repository;

    @Override
    public void doIt(throws Exception {

        repository.deleteAll();

        // save a couple of customers
        repository.save(new Customer("Alice", "Smith"));
        repository.findByName("Smith") ;

        ...
    }
}
```



MongoTemplate

A ***MongoTemplate*** bean is also auto-configured

- It is the underlying implementation of *MongoDBRepositories*
- The bean can also be used directly.



Embedded Mongo

The dependency :

`de.flapdoodle.embed:de.flapdoodle.embed.mongo`
starts an embedded MongoDB

The port used is either determined
randomly or set by the property:
`spring.data.mongodb.port`

MongoDB traces are visible, if slf4j is in
the classpath



Web Applications

Introduction to Spring MVC

Spring MVC for RESTful API
De/Serialization with Jackson
Exceptions, CORS and OpenAPI



Introduction

SpringBoot is suitable for web development

The ***spring-boot-starter-web*** starter module is used to load the Spring MVC framework

Spring MVC allows declaring beans of type

- ***@Controller*** or ***@RestController***
- Whose methods can be mapped to http requests via ***@RequestMapping***



Internal beans of Spring MVC

HandlerMapping : Associates incoming requests with handlers and defines a list of interceptors that perform pre and post processing on the request. The most common implementation and @Controller

HandlerExceptionResolver : Associates exceptions with exception handlers.

ViewResolver : Resolves from an outcome the view to render

LocaleResolver & ***LocaleContextResolver*** : Determines the locale of the client

ThemeResolver : Resolves the theme to use for the application (layout and style)

MultipartResolver : Processes multi-part requests for file loading for example.

FlashMapManager : Allows request attributes to be transferred from one request to another when a redirection occurs



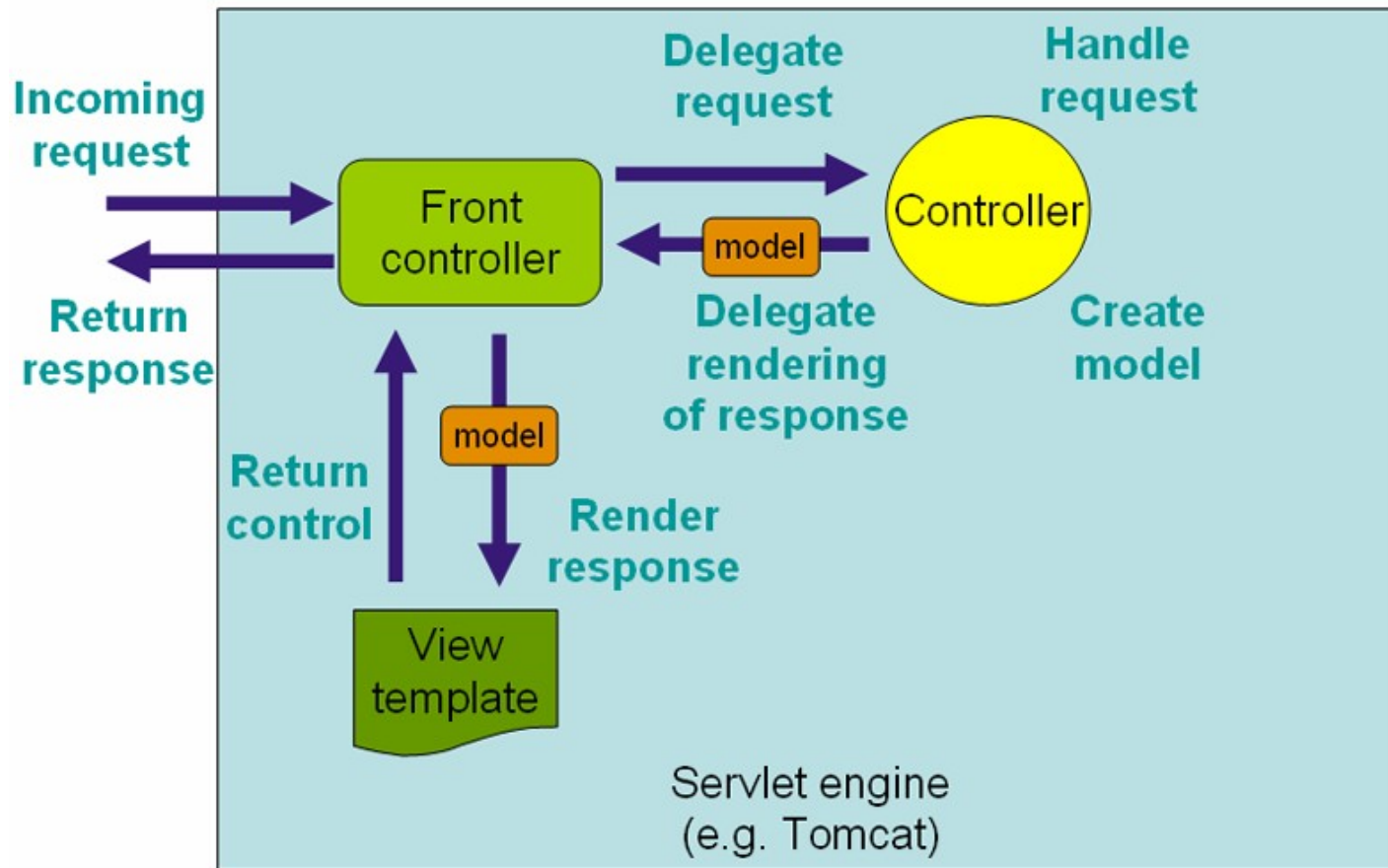
MVC Model

The MVC framework is build around the ***DispatcherServlet*** which dispatch HTTP request to controllers

The controller/request association is done through ***@RequestMapping*** annotations

- Classic controllers are responsible for preparing model data through ***Map-like*** interfaces.
The request processing is then transferred to a rendering technology (JSP, Velocity, Freemarker, Thymeleaf) which selects a page template and generates HTML
- REST controllers are responsible for constructing an HTTP response (return code, headers, etc.) whose body is generally a ***json*** document

Classical MVC





Example of Web Controller

@Controller

@RequestMapping("/web")

```
public class MembersController {  
    @Autowired  
    protected MemberRepository memberRepository;
```

@GetMapping(path = "/register")

```
public String registerForm(Model model) {  
    model.addAttribute("user", new User());  
    return "register";  
}
```

@RequestMapping(path = "/register", method = RequestMethod.POST)

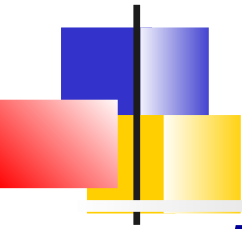
```
public String register(@Valid @RequestBody Member member) {  
    member = memberRepository.save(member);  
    return "home" ;  
}  
}
```



Other features

Other features are provided by the framework:

- Location, time zone
- Theme resolution
- Support for downloading files
- ...



Thymeleaf

Thymeleaf is the recommended view technology

It allows to build HTML-5 pages

Views consist of:

- HTML-5 tags
- Thymeleaf attributes ***data-th-*** (or shorthand ***th-***)
- Expressions that produce dynamic data



Attributes

th:text : Allows externalization (and localization of texts)

th:object : Selecting a context variable

th:errors : Access to error messages

th:href, th:attr, th:action : Allows to resolve URLs, tag attributes, form actions

th:insert, th:replace, th:with : Inclusion of fragments

th:if, th-each : Test, loops

th:span, th:div, th:class, ... : Tags and CSS

th:on* : Javascript Events

....



Thymeleaf expressions

- `${...}`** : Access to the context
- `*{...}`** : Relative to the current variable selected via `th:object`
- `#{...}`** : Localized messages
- `@{...}`** : URL
- `~{...}`** : HTML Fragment `<=>` inclusion



Example Thymeleaf

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org"
      xmlns:sec="http://www.thymeleaf.org/thymeleaf-extras-springsecurity3">
  <head><title>Registration Form </title></head>
  <body>
    <form th:action="@{/web/register}" th:object="${user}" method="post">
      <div><label> First name : <input type="text" th:field="*{firstName}"/> </label></div>
      <div><label> Last name: <input type="text" th:field="*{lastName}"/> </label></div>
      <div><input type="submit" value="Sign In"/></div>
      <div><label> Email : <input type="text" th:field="*{email}"/> </label></div>
      <div><label> Password: <input type="password" th:field="*{password}"/> </label></div>
      <div><input type="submit" value="Sign In"/></div>
    </form>
  </body>
</html>
```



Auto-configuration

SpringBoot performs automatic configurations for Spring MVC. The main contributions are:

- Automatic start of embedded servers
- Default configuration to serve static resources (index.html, favicon, Webjars)
- Automatic detection and configuration of the templating language
- Automatic configuration of *HttpMessageConverters* allowing default behavior of serializers



Web Applications

Introduction to Spring MVC
Spring MVC for RESTful API
De/Serialization with Jackson
Exceptions, CORS and OpenAPI



@RestController

The **@RestController** annotation is specified on simple classes whose public methods are generally accessible via HTTP

@RestController

@RequestMapping("/api")

```
public class HelloWorldController {  
  
    @GetMapping("/helloWorld")  
    public String helloWorld() {  
        return "helloWorld";  
    }  
}
```



@RequestMapping

@RequestMapping

- At the class level, it indicates that all methods of the controller will be relative to this path
- At a method level, the annotation specifies the following attributes:

- **path**: Fixed value or URI mask
- **method**: To limit the method to an HTTP action
- **produce/consume**: Specify input/output data format

In the context of a Rest API, it is not necessary to specify these attributes because the format is always JSON



Variants of *@RequestMapping*

Variants exist to limit to a single method.
These are the annotations that are
generally used in a Rest API:

@GetMapping,
@PostMapping,
@PutMapping,
@DeleteMapping,
@PatchMapping



Method Argument Types

A method annotated via *@*Mapping* can ask Spring to inject some arguments. The possible types are :

- The HTTP request or response (*ServletRequest*, *HttpServletRequest*, *spring.WebRequest*, ...)
- The HTTP session (*HttpSession*)
- The locale, the time zone
- The HTTP method
- The user authenticated by HTTP (Primary)
- ...

If the argument is of another type (Domain model class or basic types), it requires **annotations** so that Spring can perform the necessary conversions from the HTTP request



Method Argument Annotations

These annotations are used to associate an argument with a value of the HTTP request.

The main annotations used as part of a Rest API are

- **@PathVariable**: Part of the URI
- **@RequestParam**: An HTTP parameter (usually passed by the character ?)
- **@RequestBody**: Request content in Json format that will be converted to a Java object
- **@RequestHeader**: An HTTP header
- **@RequestPart**: Part of a multi-part request



@PathVariable and URI templates

A URI template allows you to define placeholders.

Ex ::

<http://www.example.com/users/{userId}>

The ***@PathVariable*** annotation associates the placeholder with a method argument that has the same name

```
@GetMapping("/owners/{ownerId}")  
public String findOwner(@PathVariable String ownerId) {
```



HTTP Parameters with *@RequestParam*

```
@RestController
@RequestMapping("/pets")
public class PetController {

    // ...

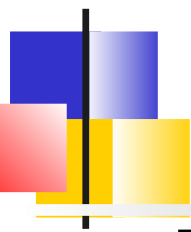
    @GetMapping
    public Pet getPet(@RequestParam("petId") int petId) {
        Pet pet = this.clinic.loadPet(petId);

        return pet;
    }

    // ...

}
```

=> Example ***http://<server>/pets?petId=5***



@RequestBody and converter

The *@RequestBody* annotation converts the JSON body of the request into an object of the domain.

- It is typically used on annotated methods *@PostMapping*, *@PutMapping*, *@PatchMapping*
- The conversion, called de-serialization, is performed by the Jackson library



Example *@RequestBody*

```
@RestController
@RequestMapping("/pets")
public class PetController {

    // ...

    @PostMapping
    public Pet savePet(@RequestBody Pet pet) {
        Pet pet = this.clinic.savePet(pet);

        return pet;
    }

    // ...

}
```



Types of return values

The possible types of return values for a REST controller are:

- A **Model** or **DTO** class that will be converted into JSON via the Jackson library.
The return status code is then 200
- A **ResponseEntity<T>** object allowing to specify the return status codes and HTTP headers



Examples

```
@RestController
@RequestMapping(value="/users")
public class MyRestController {

    @GetMapping(value="/{user}")
    public User getUser(@PathVariable Long user) {
        // ...
    }

    @GetMapping(value="/{user}/customers")
    List<Customer> getUserCustomers(@PathVariable Long user) {
        // ...
    }

    @DeleteMapping(value="/{user}")
    public ResponseEntity<Void> deleteUser(@PathVariable Long user) {
        // ...
        return new ResponseEntity<>(HttpStatus.ACCEPTED);
    }

    @PostMapping
    public ResponseEntity<User> register(@RequestBody User user) {

        user = userRepository.save(user);

        return new ResponseEntity<>(user, HttpStatus.CREATED);
    }
}
```



Web Applications

Introduction to Spring MVC
Spring MVC for RESTful API
De/Serialization with Jackson
Exceptions, CORS and OpenAPI



JSON serialization

One of the main issues with RestFul APIs is the conversion of domain objects to JSON format.

Specialized libraries are used (Jackson, Gson), they allow to benefit from default behavior

But, usually the developer has to fix some issues:

- Infinite loop for bidirectional relationships between model classes
- Adaptation to the needs of the front-end interface
- Optimization of the volume of data exchanged
- Date format



Default behavior

```
public class Member {  
    private long id;  
    private String nom,prenom;  
    private int age;  
    private Date registeredDate;  
}
```

Becomes :

```
{  
    "id": 5,  
    "nom": "Dupont",  
    "prenom": "Gaston",  
    "age": 71,  
    "registeredDate": 1645271583944 // Nombre de ms depuis le 1er Janvier 1970  
}
```



Jackson Concepts

With Jackson, serializations/deserializations are done by ***ObjectMapper***

// Sérialization

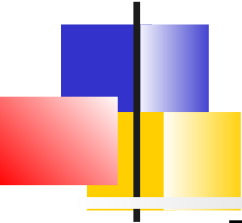
```
Member m = memberRepository.findById(4l) ;  
ObjectMapper objectMapper = new ObjectMapper() ;  
String jsonString = ObjectMapper.writeValueAsString(m) ;
```

...

// Désérialization

```
String jsonString= "{\n\"id\" : 5,\n\" + ... + \"}\" ;  
Member m2 = ObjectMapper.readValue(jsonString) ;
```

In a SpringBoot context, we rarely use the *ObjectMapper* object directly ... but we influence its behavior through annotations.



Solutions to serialization issues

To adapt Jackson's default serialization to the requirements, 4 approaches:

- Create **specific DTO** classes.
- Use the **annotations proposed by Jackson** to control the serialization of a domain class
- Use **@JsonView** annotation. To adapt the serialization of a model class to a use case
- Implement your **own Serializer/Deserializer**.
To fulfill specific requirements.
Spring offers *@JsonComponent* annotation



Example DTO

```
@Service
public class UserService {
    @Autowired UserRepository userRepository;
    @Autowired RolesRepository rolesRepository;

    UserDto retrieveUser(String login) {
        User u = userRepository.findByLogin(login);
        List<Role> roles = rolesRepository.findByUser(u);

        return new UserDto(u,roles);
    }
}
```

```
public class UserDto {
    private String login, email, nom, prenom;
    List<Role> roles;

    public UserDto(User user, List<Role> roles) {
        login = user.getLogin(); email = user.getEmail();
        nom = user.getNom(); prenom = user.getPrenom();
        this.roles = roles;
    }
}
```



Two-way relationships

The problem

```
public class User {  
    public int id;  
    public String name;  
    public List<Item> userItems;  
}  
  
public class Item {  
    public int id;  
    public String itemName;  
    public User owner;  
}
```

When Jackson serializes one of the 2 classes, it falls into an infinite loop



Two-way relationships

A solution

Annotating the 2 classes with **@JsonManagedReference** and **@JsonBackReference**

```
public class User {  
    public int id;  
    public String name;  
  
    @JsonManagedReference  
    public List<Item> userItems;  
}
```

```
public class Item {  
    public int id;  
    public String itemName;  
  
    @JsonBackReference  
    public User owner;  
}
```

The *userItems* property is serialized but not *owner*



Two-way relationships

Another solution

Annotating classes with **@JsonIdentityInfo** which instructs Jackson to serialize a class just with its ID

```
@JsonIdentityInfo(  
    generator = ObjectIdGenerators.PropertyGenerator.class, property = "id")  
public class User {...}
```

```
@JsonIdentityInfo(  
    generator = ObjectIdGenerators.PropertyGenerator.class, property = "id")  
public class Item { ... }
```

Serializing an Item:

```
{  
  "id":2,  
  "itemName":"book",  
  "owner":  
    {  
      "id":1,  
      "name":"John",  
      "userItems":[2]  
    }  
}
```



Two-way relationships

Still another solution

Annotating classes with **@JsonIgnore** tells Jackson not to serialize a property

```
public class User {  
    public int id;  
    public String name;  
  
    public List<Item> userItems;  
}
```

```
public class Item {  
    public int id;  
    public String itemName;  
  
    @JsonIgnore  
    public User owner;  
}
```




@JsonView

Empty static class with inheritance relationships can be defined. They represent a certain way for de/serializing

```
public class CompanyViews {  
    public static class Normal{};  
    public static class Manager extends Normal{};  
    public static class HR extends Normal{};  
}
```

The classes are then referenced via the *@JsonView* annotation:

- On model classes:

Which attribute is serialized when such view is enabled?

- On controller methods:

Which view should be used when serializing the return value of this method?



Refrencing views in the model

```
..  
public class Staff {  
  
    @JsonView(CompanyViews.Normal.class)  
    private String name;  
  
    @JsonView(CompanyViews.Normal.class)  
    private int age;  
  
    // 2 vues  
    @JsonView({CompanyViews.HR.class, CompanyViews.Manager.class})  
    private String[] position;  
  
    @JsonView(CompanyViews.Manager.class)  
    private List<String> skills;  
  
    @JsonView(CompanyViews.HR.class)  
    private Map<String, BigDecimal> salary;
```



Activating a view in the controller

```
@RestController
public class StaffController {

    @GetMapping
    @JsonView(CompanyViews.Normal.class)
    public List<Staff> findAll() {
    }

    ...

    ObjectMapper mapper = new ObjectMapper();

    Staff staff = createStaff();

    try {
        String normalView =
            mapper.writerWithView(CompanyViews.Normal.class).writeValueAsString(staff);
```



Other Jackson annotations

***@JsonProperty, @JsonGetter,
@JsonSetter, @JsonAnyGetter,
@JsonAnySetter, @JsonIgnore,
@JsonIgnoreProperty, @JsonIgnoreType :***
Allowing to set JSON properties

@JsonRootName : Noeud racine

@JsonSerialize, @JsonDeserialize :
Indicates specialized de/serializers

....



Date Format

To have a String representation of the dates according to the wishes of the front-ent, the more flexible is to use ***@JsonFormat***

```
public class Event {  
    public String name;  
    @JsonFormat(shape = JsonFormat.Shape.STRING,  
                pattern = "dd-MM-yyyy hh:mm:ss")  
    public Date eventDate;  
}
```



Specific serializer

Spring **@JsonComponent** annotation makes it easy to register Jackson serializers/deserializers

It must be placed on implementations of *JsonSerializer* and *JsonDeserializer* or on classes containing inner-class of this typ

@JsonComponent

```
public class Example {  
    public static class Serializer extends JsonSerializer<SomeObject> {  
        // ...  
    }  
    public static class Deserializer extends  
JsonDeserializer<SomeObject> {  
        // ...  
    }  
}
```



Web Applications

Introduction to Spring MVC
Spring MVC for RESTful API
De/Serialization with Jackson
Exceptions, CORS and OpenAPI



Spring MVC configuration customization

Customizing SpringBoot's default configuration can be done by defining a bean of type ***WebMvcConfigurer*** and overriding the proposed methods.

- For Rest API, a method allows you to configure the CORS¹

1. CORS : *Cross-origin resource sharing*, a web page cannot make requests to servers other than its origin server.



Example Cross-origin

CORS can be configured globally by overriding the ***addCorsMapping*** method of *WebMvcConfigurer*:

```
@Configuration
public class MyConfiguration implements WebMvcConfigurer {
    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/api/**").allowedOrigins("*");
    }
}
```

Note that it is also possible to configure the cors individually on the controllers via the ***@CrossOrigin*** annotation



Error management

Spring Boot associates ***/error*** with the global application error page

- A default behavior in REST or in Web allows to visualize the cause of the error

To override the default behavior in a context of RESTful APIs:

- The ***ResponseStatus*** annotation on a business exception thrown by a controller
- Use the ***ResponseStatusException*** class to associate a return code with an Exception
- Add a class annotated by ***@ControllerAdvice*** to centralize response generation during exceptions



Example

@ResponseStatus(value = HttpStatus.NOT_FOUND)

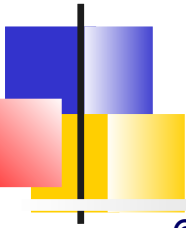
```
public class MyResourceNotFoundException extends RuntimeException {  
    public MyResourceNotFoundException() {  
        super();  
    }  
    public MyResourceNotFoundException(String message, Throwable cause) {  
        super(message, cause);  
    }  
    public MyResourceNotFoundException(String message) {  
        super(message);  
    }  
    public MyResourceNotFoundException(Throwable cause) {  
        super(cause);  
    }  
}
```



ResponseStatusException

```
@GetMapping(value =("/{id}")
public Foo findById(@PathVariable("id") Long id, HttpServletResponse response)
{
    try {
        Foo resourceById = RestPreconditions.checkFound(service.findOne(id));

        eventPublisher.publishEvent(new SingleResourceRetrievedEvent(this,
response));
        return resourceById;
    }
    catch (MyResourceNotFoundException exc) {
        throw new ResponseStatusException(
            HttpStatus.NOT_FOUND, "Foo Not Found", exc);
    }
}
```



Example *@ControllerAdvice*

@ControllerAdvice

```
public class NotFoundAdvice extends ResponseEntityExceptionHandler {

    @ExceptionHandler(value = {MemberNotFound.class, DocumentNotFound.class})
    ResponseEntity<Object> handleNotFoundException(HttpServletRequest request,
        Throwable ex) {
        return new ResponseEntity<Object>(
            "Entity was not found", new HttpHeaders(), HttpStatus.NOT_FOUND);
    }

    @Override
    protected ResponseEntity<Object>
        handleMethodArgumentNotValid(MethodArgumentNotValidException ex,
            HttpHeaders headers, HttpStatus status, WebRequest request) {
        return new ResponseEntity<Object>(
            ex.getMessage(), new HttpHeaders(), HttpStatus.BAD_REQUEST);
    }
}
```



SpringDoc

SpringDoc is a tool that simplifies the generation and maintenance of REST API documentation

It is based on the OpenAPI 3 specification and integrates with Swagger-UI

Just put the dependency in the build file:

```
<dependency>
  <groupId>org.springdoc</groupId>
  <artifactId>springdoc-openapi-ui</artifactId>
  <!-- OU : springdoc-openapi-webflux-ui -->
  <version>1.5.2</version>
</dependency>
```



Features

By default,

- The OpenAPI description is available at:
<http://localhost:8080/v3/api-docs/>
- The Swagger interface to:
<http://localhost:8080/swagger-ui.html>

SpringDoc takes into account

- *javax.validation* annotations positioned on DTOs
- Exceptions handled by *@ControllerAdvice*
- OpenAPI annotations
<https://javadoc.io/doc/io.swagger.core.v3/swagger-annotations/latest/index.html>

SpringDoc can be disabled via property :
`springdoc.api-docs.enabled=false`



OpenInView

In a web application, Spring Boot registers the *OpenEntityManagerInViewInterceptor* interceptor by default in order to apply the “***Open EntityManager in View***” pattern to avoid *LazyExceptions* in views

If this is not the desired behavior:
`spring.jpa.open-in-view = false`



Spring Security

Principles

Stateful and stateless models
Autoconfiguration Spring Boot
oAuth2



Spring Security

Spring Security provides support for 2 aspects of security:

- **Authentication** : Ensure the identity of a user or a system
- **Authorization** : Check that the user or the system has permission to access to a resource.

Support consist of :

- Integration of standard authentication techniques (LDAP, OpenID, Kerberos, PAM, CAS, OAuth2) or completely custom
- Declarative ACLs constraint on the web and business layer



Principles and mechanisms

On the web layer, security is implemented by a chain of filters referenced in the bean ***springSecurityFilterChain*** .

This bean is highly configurable and can be adapted to many different needs. Configuration is made by Java code

Security on the business layer is not enabled by default. To enable it, we must set ***@EnableGlobalMethodSecurity***

Security is then implemented through interceptors, and declarative ACLs can be set on business method with annotations



Some common filters of *springSecurityFilterChain*

UsernamePasswordAuthenticationFilter :

Responds to */login* by default, retrieves username and password parameters, and invokes the authentication handler

SessionManagementFilter : Management of collaboration between http session and security

BasicAuthenticationFilter : Process basic authentication authorization headers

SecurityContextPersistenceFilter : Responsible for storing security context



Customization of the web layer security

Customization consists of implementing a class of type ***WebSecurityConfigurer*** and overriding the default behavior by overriding the appropriate methods. Specifically :

- ***void configure(HttpSecurity http)*** : Allows to define ACLs and filters of springSecurityFilterChain
- *Authentication: 2 alternatives*
 - ***void configure(AuthenticationManagerBuilder auth)*** : Allows to build the authentication manager which can be provided by Spring (inMemory, jdbc, ldap, ...) or completely personalized by the implementation of a bean ***UserDetailsService***
 - ***AuthenticationManager authenticationManagerBean()*** : Creating a Bean Implementing Password Verification
- ***void configure(WebSecurity web)*** : Allows to define the behavior of the filter when requesting static resources



Example : Configuring the filter chain

```
protected void configure(HttpSecurity http) throws Exception {
    http
        .authorizeRequests() // ACLs
        .antMatchers("/resources/**", "/signup", "/about").permitAll()
        .antMatchers("/admin/**").hasRole("ADMIN")
        .antMatchers("/db/**").access("hasRole('ADMIN') and hasRole('DBA')")
        .anyRequest().authenticated()
        .and()
        .formLogin() // Page de login
        .loginPage("/login")
        .permitAll()
        .and()
        .logout() // Comportement du logout
        .logoutUrl("/my/logout")
        .logoutSuccessUrl("/my/index")
        .invalidateHttpSession(true)
        .addLogoutHandler(logoutHandler)
        .deleteCookies(cookieNamesToClear) ;
}
```



Definition of filters

The order of the filters is very important.

A priori, the methods accessible on *HttpSecurity* make it possible to activate the filters without worrying about the order in which they will be activated.

To modify the sequence of filters at a lower level, it is possible to directly use the methods ***addFilter****



Debug security

To debug security :

- At startup Spring log on the console the constitution of the *springSecurityFilterChain*

```
INFO [mo.s.s.web.DefaultSecurityFilterChain : Will secure ... with []
```

To debug filter executions:

- Activate :

```
logging.level.org.springframework.security=DEBUG
```




Example : Authentication Manager Configuration

```
@Configuration
```

```
public class InMemorySecurityConfiguration extends  
    WebSecurityConfigurerAdapter {
```

```
@Override
```

```
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
```

```
    // Here we use a memory realm, AuthenticationManagerBuilder allows  
    // also easy to connect to an LDAP directory or a database
```

```
    auth.inMemoryAuthentication().withUser("user").password("password").  
        roles("USER")  
        .and().withUser("admin").password("password").  
        roles("USER", "ADMIN");
```

```
    }  
}
```



Customization via *UserDetailsService*

An alternative to customizing authentication is to provide a bean implementing ***UserDetailsService***

The interface contains a single method :

```
public UserDetails loadUserByUsername(String login) throws  
UsernameNotFoundException
```

- It is responsible for returning, from a login, an object of type `UserDetails` encapsulating the password and the roles
It is the framework that checks if the entered password matches.
- The presence of a *UserDetailsService* type bean is sufficient for its configuration



Example

```
import org.springframework.security.core.userdetails.User ;
...
@Service
public class UserDetailsServiceImpl implements UserDetailsService{
    @Autowired
    private AccountRepository accountRepository;

    @Transactional(readOnly = true)
    public UserDetails loadUserByUsername(String login) throws UsernameNotFoundException {
        Account account = accountRepository.findByLogin(login);
        if ( account == null )
            throw new UsernameNotFoundException("Invalides login/mot de passe");
        Set<GrantedAuthority> grantedAuthorities = new HashSet<>();
        for (Role role : account.getRoles()){
            grantedAuthorities.add(new SimpleGrantedAuthority(role.getLibelle()));
        }
        return new User(account.getLogin(), account.getPassword(), grantedAuthorities);
    }
}
```



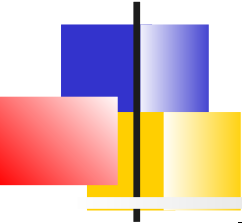
Password Encoder

Spring Security 5 requires passwords to be encrypted

It is then necessary to define a bean of type ***PasswordEncoder***

The recommended implementation is *BcryptPasswordEncoder*

```
@Bean
PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
}
```



{noop}

If the passwords are stored in plain text, they must be prefixed with ***{noop}*** so that Spring Security does not use an encoder

```
public UserDetails loadUserByUsername(String login) throws UsernameNotFoundException {
    Member member = memberRepository.findByEmail(login);
    if ( member == null )
        throw new UsernameNotFoundException("Invalides login/mot de passe");
    Set<GrantedAuthority> grantedAuthorities = new HashSet<>();

    return new User(member.getEmail(), "{noop}" + member.getPassword(), grantedAuthorities);
}
```



Successful authentication

If authentication is successful, the ***Authentication*** object is stored in a *ThreadLocal*.

The object can be retrieved by any class via the static method :

SecurityContextHolder.getContext().getAuthentication()

Authentication object encapsulates user info and his granted authorities



Spring Security

Principles

Stateful and stateless models

Autoconfiguration Spring Boot

oAuth2



Web App and Rest API

Web applications (stateful) and REST APIs (stateless) do not generally have the same strategy for security management.

- In a stateful application, information related to authentication is stored in the user session (cookie).
- In a stateless application, user rights are passed on each request via a token



Web app authentication process

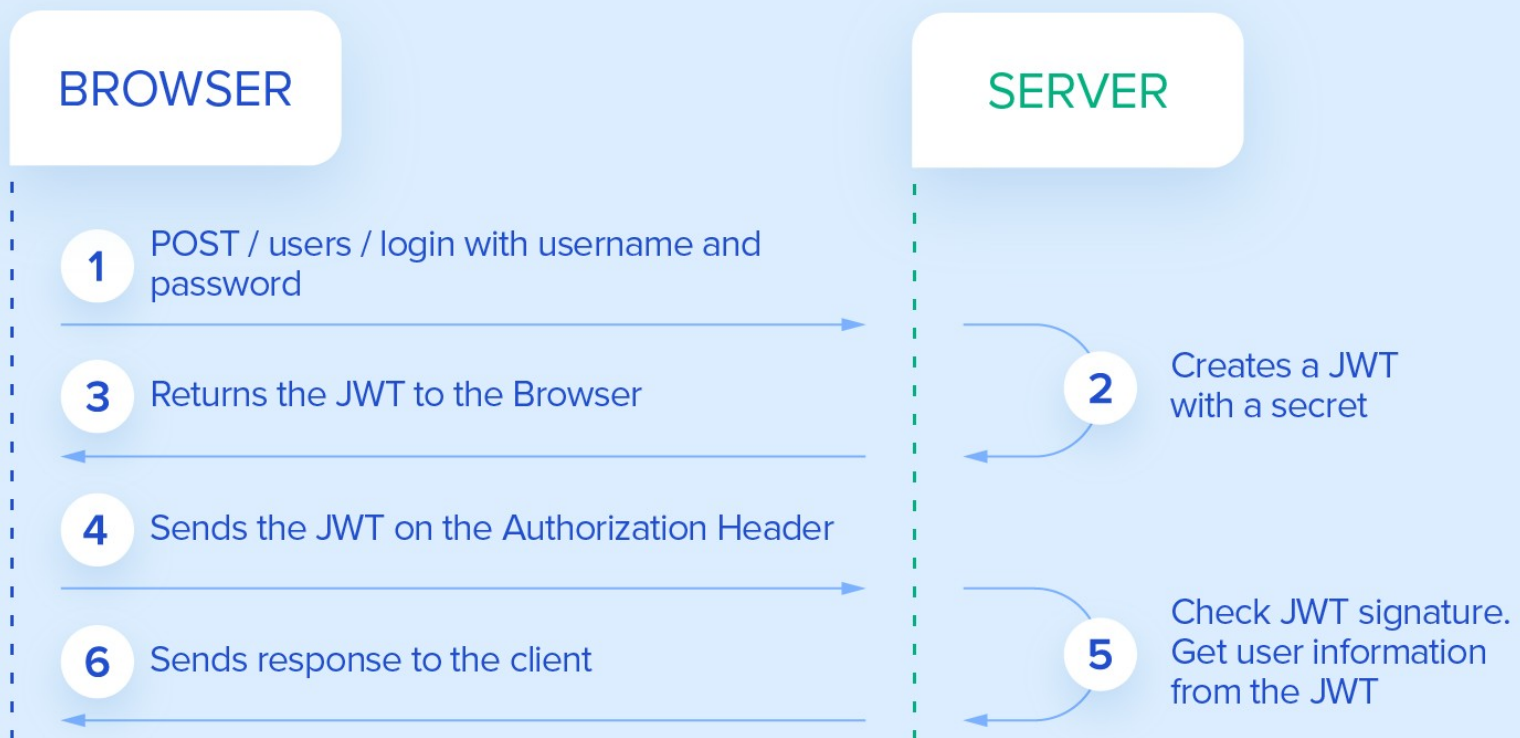
1. The client requests a protected resource.
2. The server returns a response indicating that one must authenticate:
 - 1.By redirecting to a login page
 - 2.By providing headers for basic browser authentication.
- 3.The browser returns a response to the server:
 - 1.Either the POST of the login page
 - 2.Either the authentication HTTP headers.
4. The server decides if the credentials are valid:
 - 1.if yes. Authentication is stored in the session, the original request is retried, if the rights are sufficient the page is returned otherwise a 403 code
 - 2.If not, the server asks for authentication again.
5. The Authentication object containing the user and his roles is present in the session.



REST API authentication process

1. The client requests a protected resource.
2. The server returns a response indicating that URL needs authentication by sending a 403 response.
3. The browser offers a login form then sends the form to an authentication server (may be different than the API server)
4. The authentication server decides if the credentials are valid:
 1. if yes. It generates a token with a validity period
 2. If not, the server asks for authentication again.
5. The client retrieves the token and associates it with all requests to the API
6. The resource server decodes the token and derives the user's rights. It authorizes or prohibits access to the resource.
Authentication object is set

Authentication Rest





Spring Security

Principles

Stateful and stateless models

Autoconfiguration Spring Boot

oAuth2



Auto-configuration

If Spring Security is in the classpath, the default configuration is :

- All URLs of the web application are secured by form authentication
- A simplistic authentication manager is configured to allow the identification of a single user
 - The generated password is printed on the console



Default Authentication Manager

The default authentication manager defines a single user user with a random password that is displayed on the console at startup.

Default can be changed via

```
security.user.name= myUser  
security.user.password=secret
```



Other default features

Other features are automatically obtained:

- Paths for standard static resources are ignored (/css/**, /js/**, /images/**, /webjars/** and **/favicon.ico).
- Security related events are published to *ApplicationEventPublisher* via *DefaultAuthenticationEventPublisher*
- Common low-level features (HSTS, XSS, CSRF, caching)
- Login and logout forms



TLS / SSL

TLS / SSL can be configured via properties prefixed with ***server.ssl.****

For example :

```
server.port=8443
server.ssl.key-store=classpath:keystore.jks
server.ssl.key-store-password=secret
server.ssl.key-password=another-secret
```

By default if SSL is configured, port 8080 disappears.
If you want both, you must explicitly configure the network connector in the web server



Spring Security

Principles

Stateful and stateless models
Autoconfiguration Spring Boot
oAuth2



Introduction

oAuth2 is a protocol between different actors that allows to specify authorization constraints on protected resources

It is also used via OpenID Connect for authentication

Spring Security offers support to:

- Authenticate on an external OpenID authorization server (Github, Google, Facebook, ...)
- Protect resource URLs via *oAuth2*
- Integrate with the Okta solution



Roles of protocol

The **Client** is the application trying to access the user account. It needs to get permission from the user to do so.

The **resource server** is the API with protected resources

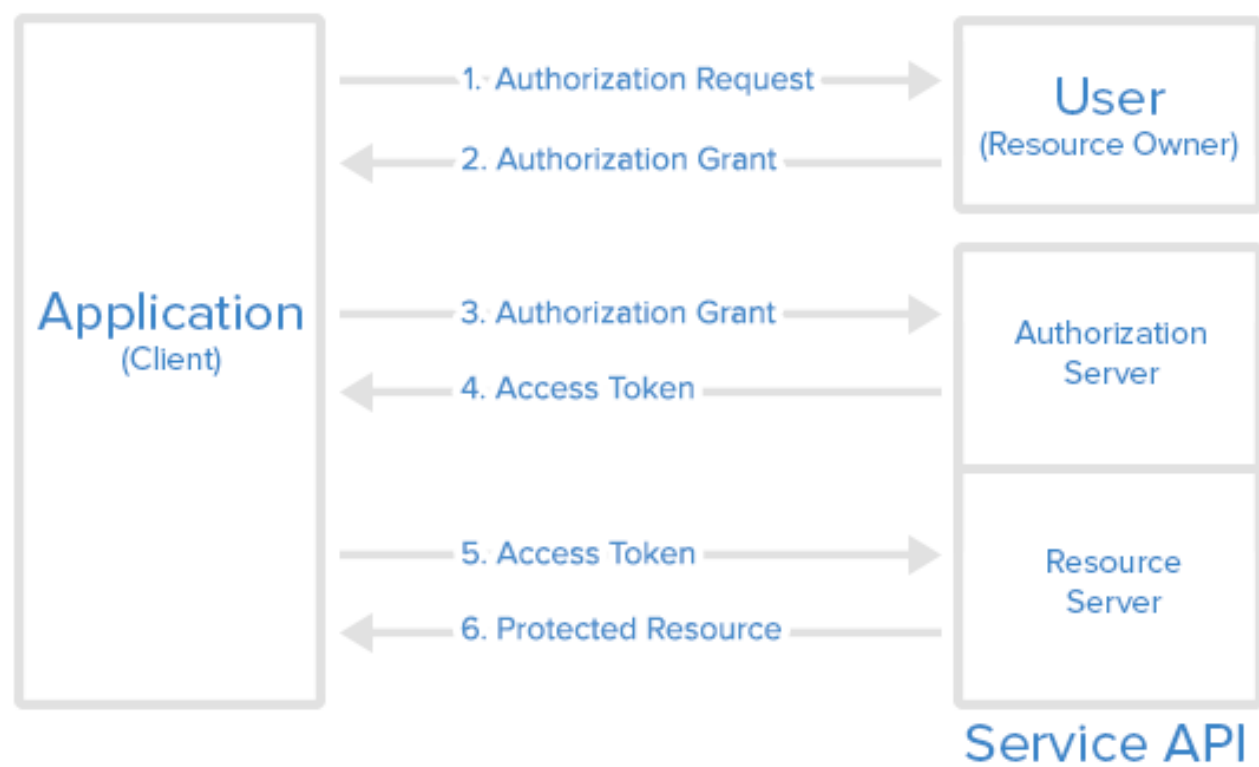
The **authorization server** is the server that authorizes a client to access resources by providing a token. It can request user approval to do so

The **user** is the person who gives access to certain parts of his account

NB: A protocol participant can play multiple roles



Abstract Protocol Flow





Scenario

1. Pre-register the client with the authorization service (=> client ID and a secret)
2. Obtain user grant (Different types of grants)
3. Obtaining the token (expiration date)
4. Access the protected resource using the token
5. Validation of the token by the resource server



Tokens

Tokens are random character strings generated by the authorization server

Tokens are then transmitted in HTTP requests and contain sensitive information => HTTPS

There are 2 types of tokens

- The **access token**: It has a limited lifespan.
- The **Refresh Token**: Issued with the access token. It is sent back to the authorization server to renew the access token when it has expired



Token usage

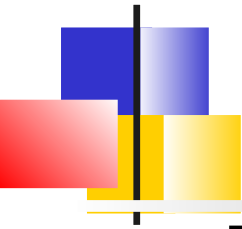
Token is generally provided via the
Authorization HTTP header

```
GET /profile HTTP/1.1
```

```
Host: api.example.com
```

```
Authorization: Bearer MzJmNDc3M2VjMmQzN
```

<http://www.bubblecode.net/en/2016/01/22/understanding-oauth2/>



Scopes

The ***scope*** is a parameter used to limit the access rights of a client

The authorization server defines the available scopes

The client can specify the scope it wants to use when accessing the authorization server



Registering the client

The protocol does not define how the client registration should be done but defines the exchange parameters.

Client must provide:

- **Application Name**: The name of the application
- **Redirect URLs**: Client URLs to receive authorization code and access token
- **Grant Types**: Types of authorizations that can be used by the client
- **Javascript Origin (optional)**: The host authorized to access resources via XMLHttpRequest

The server responds with:

- **Client Id**:
- **Client Secret**: Key to remain confidential



OAuth2 Grant Type

Different ways for the user to consent: the grant types

- ***authorization code*** :
 - The user is directed to the authorization server
 - The user consents on the authorization server
 - It is redirected to the client with an authorization code
 - The client uses the code to get the token
- ***implicit*** : Token is delivered directly. Some servers prohibit this mode
- ***password*** : The client provides the user's credentials
- ***client credentials*** : Client's credential is sufficient. Generally client is the user.
- ***device code*** :



Token validation

When receiving the token, the resource server must validate the authenticity of the token and extract its information different techniques can be evaluated :

- REST call to authorization server
- Use of shared persistent media (ex. DB)
- Use of JWT and validation via private key or public key. The most appealing



JWT

JSON Web Token (JWT) is an open standard defined in RFC 75191.

It allows the secure exchange of tokens between multiple parties.

Security consists of **verifying the integrity** of the data using a digital signature. (HMAC or RSA).

As part of a SpringBoot REST application, the token contains a user's credentials: Subject + Roles

Different implementations exist in Java (*io.jsonwebtoken, nimbus-jose-jwt...*) or the starter *spring-security-oauth2-jose*



SpringBoot's support

Support for *oAuth2* has been revised:

- The *spring-security-oauth2* project has been deprecated and replaced with SpringSecurity 5.

See :

<https://github.com/spring-projects/spring-security/wiki/OAuth-2.0-Migration-Guide>

- For the moment, there is no longer official support for an authorization server

3 starters are now available:

- ***OAuth2 Client***: Allow easy use of OpenIDConnect to log with Google, Github, Facebook, or other provider...
- ***OAuth2 Resource server***: Allow to define ACLs with client scopes and user roles contained in *oAuth2* tokens.
- ***Okta***: To work with *oAuth2* provider Okta



Solutions for an authorization server

- ♦ Use a standalone product
- ♦ A Spring project for an authorization server is in progress¹:

<https://github.com/spring-projects-experimental/spring-authorization-server>

- ♦ Another alternative is to embed an OAuth2 solution like *KeyCloak* in a SpringBoot application

See for example :

<https://www.baeldung.com/keycloak-embedded-in-spring-boot-app>



Resource server

Dependency:

```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>  
</dependency>
```

The resource server must verify the token signature to ensure that the data has not been modified.

- ***jwk-set-uri*** contains the public key that the server can use for verification
- ***issuer-uri*** points to the base authorization server URI, which can also be used to locate the endpoint providing the public key



Example *application.yml*

```
server:
  port: 8081
  servlet:
    context-path: /resource-server

spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: http://keycloak:8083/auth/realms/myRealm
          jwk-set-uri: http://keycloak:8083/auth/realms/myRealm/protocol/openid-connect/certs
```




Typical SpringBoot configuration

@Configuration

```
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.cors()
            .and()
            .authorizeRequests()
                .antMatchers(HttpMethod.GET, "/user/info", "/api/foos/**")
                    .hasAuthority("SCOPE_read")
                .antMatchers(HttpMethod.POST, "/api/foos")
                    .hasAuthority("SCOPE_write")
                .anyRequest()
                    .authenticated()
            .and()
                .oauth2ResourceServer()
                    .jwt();
    }
}
```

Voir : <https://github.com/Baeldung/spring-security-oauth.git>



Testing with SpringBoot

Spring Test

Spring Boot support
Autoconfigured tests



Versions

Spring/SpringBoot/JUnit

SpringBoot 2, Spring 5, JUnit5

First version ~2018



spring-test

Spring Test does not help much for unit testing

- Mocking of the servlet or reactive environment
- Utility Packages : *org.springframework.test.util*
- *JUnit5 integration*

But much more for integration test (tests which require an *ApplicationContext*) :

- **Cache** of the context to speed up testing
- **Injection** of class to test or test data
- **Transaction management** (roll-back after execution of one test)
- Utility classes : MockBean, MockUser
- Several mechanisms to **ease test configurations**



JUnit integration

For JUnit5, Spring provide an extension

@ExtendWith(SpringExtension.class)

This extension allow mainly to load an *ApplicationContext* before the execution of the test



Test configuration

To allow specific `ApplicationContext` during the integration test, *spring-test* provides

`@ContextConfiguration` annotation.

Main scenario is to indicate a `@Configuration` class which will be use during the test



Example JUnit5

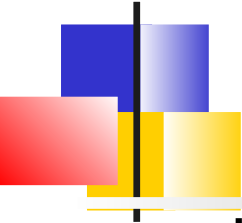
```
@ExtendWith(SpringExtension.class)
@ContextConfiguration(classes = TestConfig.class)
class SimpleTests {

    @Test
    void testMethod() {
        // test logic...
    }
}
```



Testing with SpringBoot

Spring Test
Spring Boot support
Autoconfigured tests



spring-boot-starter-test

Adding spring-boot-starter-test (in the test scope), adds the following dependencies:

- Spring Test : Spring Utilities for Testing
- **Spring Boot Test** : Utility linking Spring Test to Spring Boot
- **Spring Boot Test Autoconfigure** : Auto-configured tests
- JUnit5, AssertJ, Hamcrest
- Mockito: A framework for generating Mock classes
- JSONassert: A library for JSON assertions
- JsonPath: XPath for JSON.



Provided annotations

New annotations are available through the starter:

- **@SpringBootTest** allowing to define the Spring *ApplicationContext* to use for a test thanks to a configuration detection mechanism
- Annotations allowing self-configured tests.
Ex: Auto-configuration to test RestControllers in isolation
- Annotation for creating Mockito beans



@*SpringBootTest*

@*SpringBootTest* annotation replaces the standard *spring-test* configuration (*@ContextConfiguration* and *SpringExtension*)

- The annotation detects the application context to initialize by finding the first class annotated with *@SpringBootApplication* in the hierarchy of its package.
- Generally, it finds the main class of the application and the test is started with all the *normal* beans (~system test)



Equivalence

```
// Annotation SpringBootTest
@SpringBootTest(webEnvironment=WebEnvironment.RANDOM_PORT)
public class SpringBootTestApplicationTests {
```

```
// Classic annotations
@ExtendWith(SpringExtension.class)
@ContextConfiguration(classes =
    SprintBootTestApplication.class)
@WebAppConfiguration
public class SpringBootTestApplicationTests
```



classes attribute

The *@SpringBootTest* annotation can specify the configuration classes used to load the application context via the ***classes*** attribute

Example :

```
@SpringBootTest(classes = ForumApp.class)
```



WebEnvironment attribute

The ***WebEnvironment*** attribute lets you specify the type of application context you want:

- ***MOCK***: Provides a mocked server environment
- ***RANDOM_PORT***: The web server is started on a random port
- ***DEFINED_PORT***: The web server is started on a specified port
- ***NONE***: No servlet environment. Simple `ApplicationContext`



Mocking beans

The **@MockBean** annotation defines a Mockito bean

- This allows replacing or creating new beans
- Annotation is used on the fields of the test class, in this case the mockito bean is injected

Mockito beans are automatically reset after each test



Example *MockBean*

```
@SpringBootTest
public class MyTests {

    @MockBean
    private RemoteService remoteService;

    @Autowired
    private Reverser reverser;

    @Test
    public void exampleTest() {
        // RemoteService has been injected into the reverser bean
        given(this.remoteService.someCall()).willReturn("mock");
        String reverse = reverser.reverseSomeCall();
        assertThat(reverse).isEqualTo("kcom");
    }
}
```




Testing with SpringBoot

Spring Test
Spring Boot support
Autoconfigured tests



Auto-configured test

Spring Boot's auto-configuration capabilities may not be suitable for testing.

- When we test the controller layer, we don't want SpringBoot to automatically start a database for us

The ***spring-boot-test-autoconfigure*** module includes annotations that allow layered testing of applications



JSON Tests

In order to test if the JSON serialization is working correctly, the **@JsonTest** annotation can be used.

It automatically configures the Jackson or Gson environment

JacksonTester, *GsonTester* or *BasicJsonTester* utility classes can be injected and used, JSON-specific assertions can be used



Example

@JsonTest

```
public class MyJsonTests {

    @Autowired
    private JacksonTester<VehicleDetails> json;

    @Test
    public void testSerialize() throws Exception {
        VehicleDetails details = new VehicleDetails("Honda", "Civic");
        // Assert against a `.json` file in the same package as the test
        assertThat(this.json.write(details)).isEqualToJson("expected.json");
        // Or use JSON path based assertions
        assertThat(this.json.write(details)).hasJsonPathStringValue("@.make");
        assertThat(this.json.write(details)).extractingJsonPathStringValue("@.make")
            .isEqualTo("Honda");
    }

    @Test
    public void testDeserialize() throws Exception {
        String content = "{\"make\":\"Ford\",\"model\":\"Focus\"}";
        assertThat(this.json.parse(content))
            .isEqualTo(new VehicleDetails("Ford", "Focus"));
        assertThat(this.json.parseObject(content).getMake()).isEqualTo("Ford");
    }
}
```



MVC Controller tests

The **@WebMvcTest** annotation configures the Spring MVC framework and limits the scan to Spring MVC annotations

- It also configures *MockMvc* which allows you to do without a full Http server¹
- For *Selenium* or *HtmlUnit* testing, a web client is also provided

1. The annotation *@AutoConfigureMockMvc*, allows also to start a mock instead of a real server



Example

```
@WebMvcTest(UserVehicleController.class)
```

```
public class MyControllerTests {
```

```
    @Autowired
```

```
    private MockMvc mvc;
```

```
    @MockBean
```

```
    private UserVehicleService userVehicleService;
```

```
    @Test
```

```
    public void testExample() throws Exception {
```

```
        given(this.userVehicleService.getVehicleDetails("sboot"))
```

```
            .willReturn(new VehicleDetails("Honda", "Civic"));
```

```
        this.mvc.perform(get("/sboot/vehicle").accept(MediaType.TEXT_PLAIN))
```

```
            .andExpect(status().isOk()).andExpect(content().string("Honda
```

```
Civic"));
```

```
    }
```

```
}
```



Example (2)

```
@WebMvcTest(UserVehicleController.class)
```

```
public class MyHtmlUnitTests {
```

```
    // WebClient is auto-configured thanks to HtmlUnit
```

```
    @Autowired
```

```
    private WebClient webClient;
```

```
    @MockBean
```

```
    private UserVehicleService userVehicleService;
```

```
    @Test
```

```
    public void testExample() throws Exception {
```

```
        given(this.userVehicleService.getVehicleDetails("sboot"))
```

```
            .willReturn(new VehicleDetails("Honda", "Civic"));
```

```
        HtmlPage page = this.webClient.getPage("/sboot/vehicle.html");
```

```
        assertThat(page.getBody().getTextContent()).isEqualTo("Honda Civic");
```

```
    }
```

```
}
```



JPA Tests

@DataJpaTest configures a memory database, scans *@Entities* and configures JPA repositories

The tests are transactional and a rollback is performed at the end of the test

A *TestEntityManager* can be injected as well as a *JdbcTemplate*



Example

@DataJpaTest

```
public class ExampleRepositoryTests {  
  
    @Autowired  
    private TestEntityManager entityManager;  
  
    @Autowired  
    private UserRepository repository;  
  
    @Test  
    public void testExample() throws Exception {  
        this.entityManager.persist(new User("sboot", "1234"));  
        User user = this.repository.findByUsername("sboot");  
        assertThat(user.getUsername()).isEqualTo("sboot");  
        assertThat(user.getVin()).isEqualTo("1234");  
    }  
}
```



Other autoconfigured tests

@WebFluxTest: Testing Spring Webflux Controllers

@JdbcTest: Only datasource and jdbcTemplate.

@JooqTest: Configures a DSLContext.

@DataMongoTest: Configures a Mongo memory database, MongoTemplate, scans *@Document* classes and configures MongoDB repositories.

@DataRedisTest: Testing Redis applications.

@DataLdapTest: Embedded LDAP server (if available), LdapTemplate, *@Entry* classes and LDAP repositories

@RestClientTest: Testing REST clients. Jackson, GSON, ... + RestTemplateBuilder, and support for MockRestServiceServer.



Example

```
@RestClientTest(RestService.class)
public class RestserviceTest {
    @Autowired
    private MockRestServiceServer server;
    @Autowired
    private ObjectMapper objectMapper;
    @Autowired
    private RestService restService;

    @BeforeEach
    public void setUp() throws Exception {
        Member aMember = ...
        String memberString = objectMapper.writeValueAsString(aMember);

        this.server.expect(requestTo("/members/1"))
            .andRespond(withSuccess(memberString, MediaType.APPLICATION_JSON));
    }

    @Test
    public void whenCallingGetMember_thenOk() throws Exception {
        assertThat(restService.getMember(1)).extracting("email").isEqualTo("d@gmail.com");
    }
}
```



Test and security

Spring offers several annotations to run the tests of an application secured by SpringSecurity.

```
<dependency>
<groupId>org.springframework.security</groupId>
<artifactId>spring-security-test</artifactId>
<scope>test</scope>
</dependency>
```

@WithMockUser : The test is run with a user whose details can be specified (login, password, roles)

@WithAnonymousUser : The test is run with anonymous user

@WithUserDetails("aLogin") : The test is executed with the user loaded by *UserDetailsService*

@WithSecurityContext : Generic annotation to create whatever SecurityContext



Messaging

Support for messages brokers



Introduction

Asynchronous communications between processes provide several advantages:

- Decoupling of the producer and consumer of messages
- Scaling and load balancing
- Implementation of typical micro-services Pattern (Saga¹, Event-sourcing²)

But also difficulties

- Asynchrony management
- Setting up and operating a message broker

1. <https://microservices.io/patterns/data/saga.html>

2. <http://microservices.io/patterns/data/event-sourcing.html>

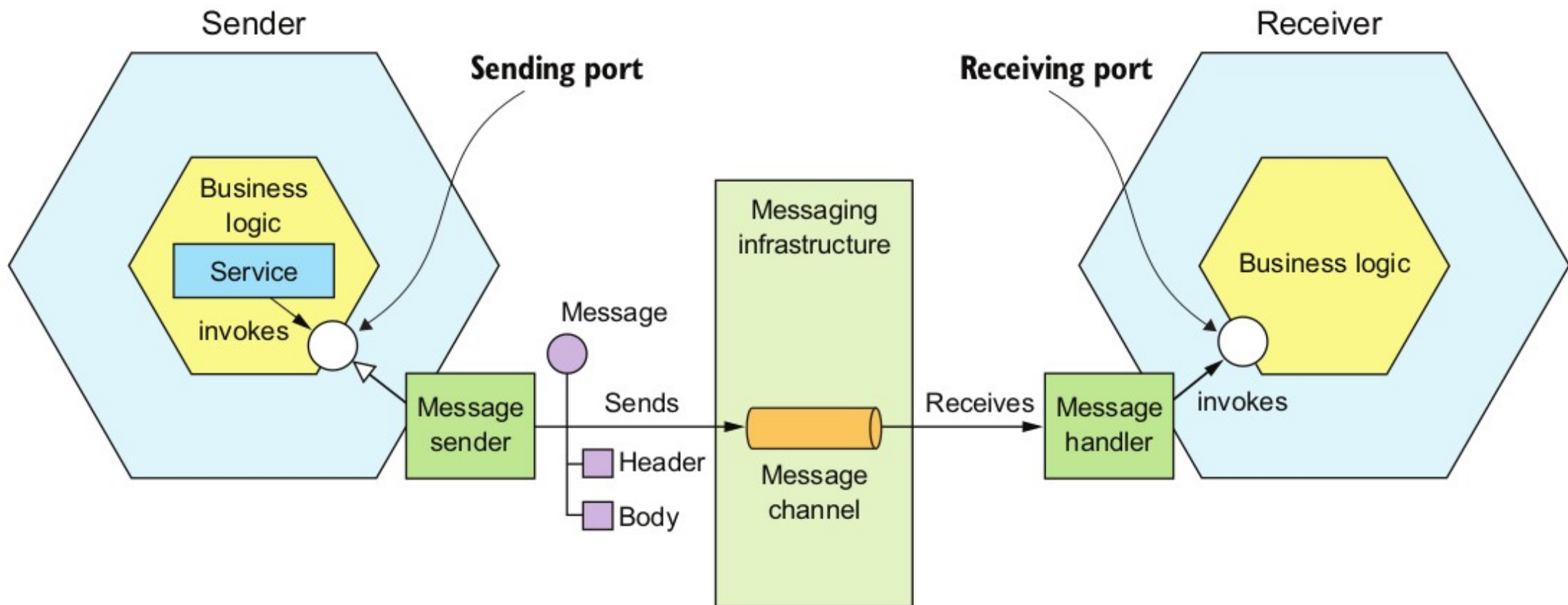


Messaging Pattern

Messaging Pattern¹: A client invokes a service using asynchronous messaging

- The messaging pattern often involves a message broker
- A client makes a request by posting an asynchronous message
- Optionally, it expects to receive a response

Architecture





Message

A message consists of headers (set of key-values) and a message body

There are 3 kinds of messages:

- **Document**: A generic message containing only data The receiver decides how to interpret it
- **Command**: A message specifying the action to invoke and its input parameters
- **Event**: A message that something just happened. Often a business event



Channels

2 kinds of channels:

- **Point-to-point** : The channel delivers the message to one of the consumers reading the channel.
Ex: Send a command message
- **PubAndSub** : The channel delivers the message to all attached consumers (subscribers)



Interaction Styles

Different interaction styles are supported:

- Synchronous Request/Response.
The client waits for the response
- Asynchronous Request/Response
The client is notified when the response arrives
- One way notification
The client does not wait for a response
- Publish and Subscribe:
The producer does not expect an answer
- Asynchronous Publish and Reply
Producer is notified when responses arrive



API Specification

Specification of an API consists of defining

- Channel names
- Types of messages and their format.
(Typically JSON)

But, unlike REST and OpenAPI, there is no standard



Message Broker

A message broker is an intermediary through which all messages transit

- The transmitter does not need to know the network location of the receiver
- The message broker buffers messages

Common implementations :

- ActiveMQ
- RabbitMQ
- Kafka
- AWS Kinesis



Spring support

Starter for pure message broker:

- RabbitMQ, ActiveMQ, Kafka, ActiveMQ Artemis, Solace PubSub

Starter for :

- Kafka Stream : Pipeline de process events

Support for *event-driven* architecture

- Spring Cloud Stream
- Spring Data Flow



Example : *spring-kafka*

Sending message

```
@Value("${app.my-channel}")
String PAYMENT_REQUEST_CHANNEL;

@Autowired
KafkaTemplate<Long, DomainEvent> kafkaOrderTemplate;

public Order doService(Domain model) {
    ...
    DomainEvent event = new DomainEvent(model);
    kafkaOrderTemplate.send(ORDER_STATUS_CHANNEL, event);
    ...
}
```

Receiving message :

```
@KafkaListener(topics = "#{ '${app.my-channel}' }", id = "oneHandler")
public void handleEvent(DomainEvent domainEvent) {
    ...
}
```



Reactive Spring

Reactive programming

Spring Reactor

Spring Data Reactive

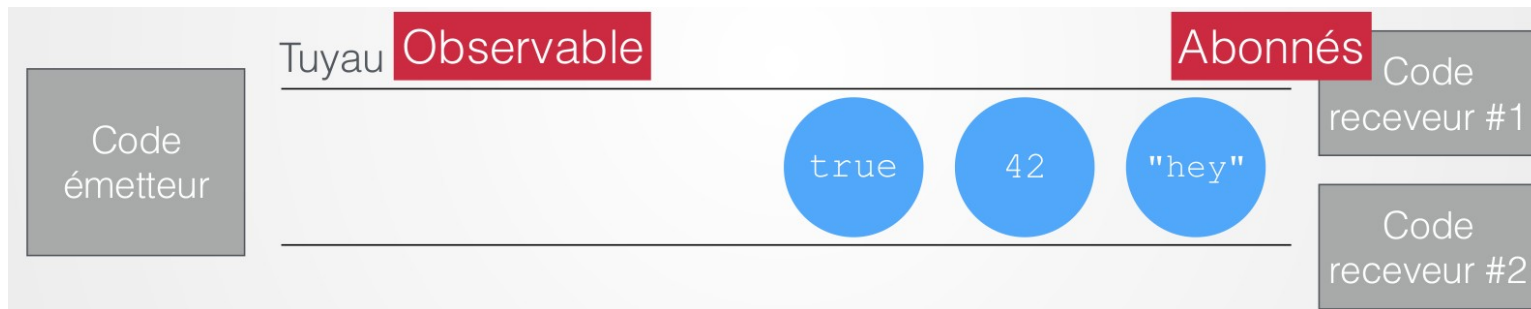
Spring Webflux



Reactive model

Consists of building your code from data streams

- Some parts of the code emit data : **Observables**
- Others react : **Subscribers** :



The sender code can send an unlimited number of values into the pipe.

The pipe can have an unlimited number of subscribers. The pipe is active as long as it has at least one follower.

Pipes are real-time feeds: as soon as a value is pushed into a pipe, subscribers receive the value and can react.



Pattern et *ReactiveX*

Reactive programming is based on the ***Observable*** pattern which is a combination of the *Observer* and *Iterator* patterns

- It uses functional programming to easily define operators on flow elements
- It is formalized by the **ReactiveX API** and many implementations exist for different languages (RxJS, RxJava, Rx.NET)



Combine and transform

Data flowing through the pipe is transformed through a series of successive operations (aka “operators”):

The operators to apply to the data are declared once and for all.

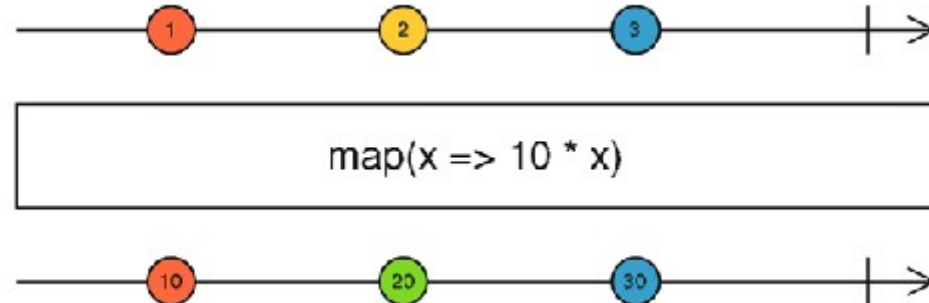
- This declarative operation is convenient to use and debug.



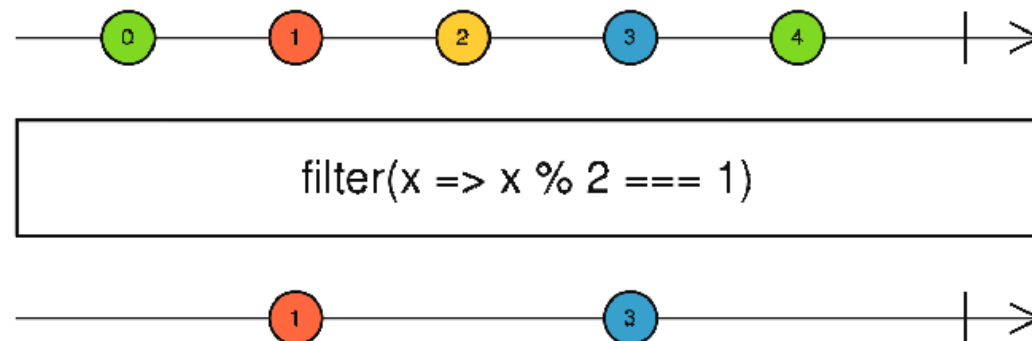
Marble diagrams

To explain the operators, the doc uses *marble diagrams*

Example *map* :



Example *filter*





Reactive Streams

Reactive Streams aims to define a standard for asynchronous processing of event streams offering **non-blocking back pressure** functionality

- It concerns Java and JavaScript environments as well as network protocols
- The standard allows interoperability but remains very low-level



Interfaces Reactive Streams

```
public interface Publisher<T> {  
    public void subscribe(Subscriber<? super T> s);  
}
```

```
public interface Subscription {  
    public void request(long n);  
    public void cancel();  
}
```

```
public interface Subscriber<T> {  
    public void onSubscribe(Subscription s);  
    public void onNext(T t);  
    public void onError(Throwable t);  
    public void onComplete();  
}
```

```
public interface Processor<T, R> extends Subscriber<T>, Publisher<R> {  
}
```



Back pressure

The concept of **back pressure** describes the possibility for subscribers to control the rate at which the events of the publishing service are sent.

- Interfaces of *Reactive Stream* support this mechanism via the method *request(int n)* in Subscription

If the Observable can't slow down, it has to make the decision to buffer, delete, or crash.



Reactor

Reactor focuses on reactive server-side programming.

It is jointly developed with Spring.

- It mainly provides 2 highest level types ***Mono*** and ***Flux*** representing a stream of events
- It offers a set of operators aligned with *ReactiveX*.
- It is an implementation of *Reactive Streams*



Reactive Spring

Reactive programming
Spring Reactor
Spring Data Reactive
Spring Webflux



Maven

```
<dependency>
```

```
  <groupId>io.projectreactor</groupId>
```

```
  <artifactId>reactor-core</artifactId>
```

```
  <version>${version}</version>
```

```
</dependency>
```



Main Types

Reactor mainly offers 2 Java types:

- ***Mono*** : Stream of 0..1 elements
- ***Flux*** : Stream of 0..N elements

Both are implementations of Reactive Stream's ***Publisher*** interface which defines 1 method:
`void subscribe(Subscriber<? super T> s)`

The stream starts transmitting only if there is a subscriber

Depending on the possible number of published events, they offer different operators



Flux

A ***Flux*** $\langle T \rangle$ represents an asynchronous sequence of 0 to N events, optionally terminated by an end signal or an error.

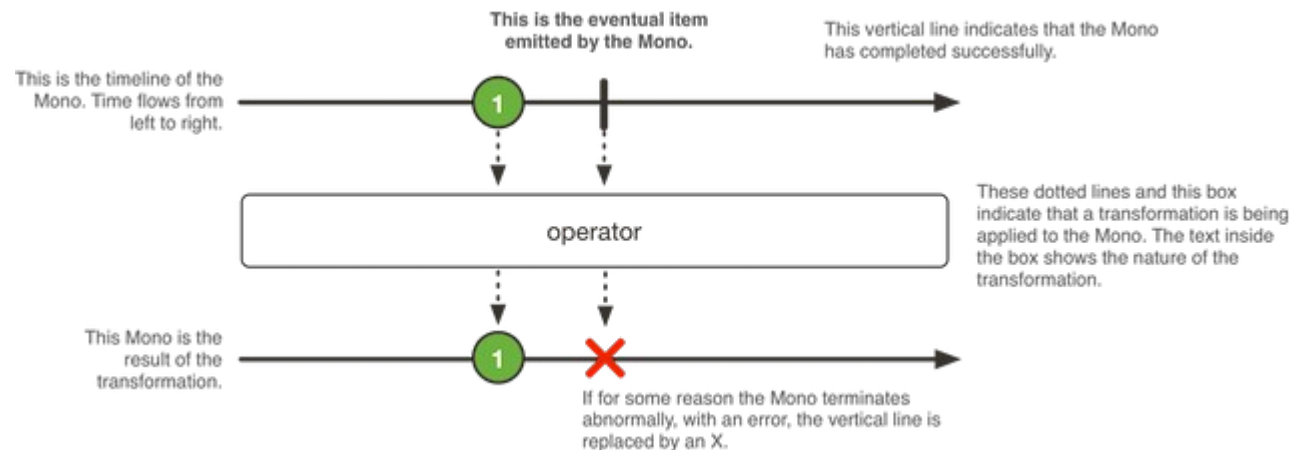
Events are translated to method calls on subscribers:

- New value : *onNext()*
- End signal : *onComplete()*
- Error : *onError()*

Mono

Mono $\langle T \rangle$ represents a sequence of 0 to 1 event, optionally terminated by an end signal or an error

Mono offers a subset of the Flux operators





Production of a data stream

The easiest way to create a *Mono* or a *Flux* is to use the available factory methods.

```
Mono<Void> m1 = Mono.empty()  
Mono<String> m2 = Mono.just("a");  
Mono<Book> m3 = Mono.fromCallable(() -> new Book());  
Mono<Book> m4 = mono.fromFuture(myCompletableFuture);
```

```
Flux<String> f1 = Flux.just("a", "b", "c");  
Flux<Integer> f2 = Flux.range(0, 10);  
Flux<Long> f3 =  
    Flux.interval(Duration.ofMillis(1000).take(10);  
Flux<String> f4 = Flux.fromIterable(bookCollection);  
Flux<Book> f5 = Flux.fromStream(bookCollection.stream());
```



Subscription

The subscription to the stream is done via the method ***subscribe()***

Generally, lambda-expressions are used

```
subscribe(); // Triggers the flow
subscribe(Consumer<? super T> consumer);
subscribe(Consumer<? super T> consumer,
          Consumer<? super Throwable> errorConsumer);
subscribe(Consumer<? super T> consumer,
          Consumer<? super Throwable> errorConsumer,
          Runnable completeConsumer);
// Chaining
subscribe(Consumer<? super T> consumer,
          Consumer<? super Throwable> errorConsumer,
          Runnable completeConsumer,
          Consumer<? super Subscription> subscriptionConsumer);
```



Subscriber Interface

Without using lambda-expressions, one can provide an implementation of the ***Subscriber*** interface that defines 4 methods:

```
void onComplete()  
void onError(java.lang.Throwable t)  
void onNext(T t)  
void onSubscribe(Subscription s)
```

Invoked after

Publisher.subscribe(Subscriber)



Subscription

Subscription represents a subscription of a (single) subscriber to a Publisher.

It is used

- to request emission events
`void request(long n)`
- To cancel the request and allow the resource to be released
`void cancel()`



Example

```
Flux.just(1, 2, 3, 4)
    .log()
    .subscribe(new Subscriber<Integer>() {
        @Override
        public void onSubscribe(Subscription s) {
            s.request(Long.MAX_VALUE); // Cause all events to be emitted
        }

        @Override
        public void onNext(Integer integer) {
            elements.add(integer);
        }

        @Override
        public void onError(Throwable t) {}

        @Override
        public void onComplete() {}
    });
```



Operators

Operators allow different types of operations on elements of the sequence:

- Transform
- Choose events
- Filter
- Handle errors
- Temporal operators
- Separate a stream
- Return to synchronous mode



Transformation

1 to 1 :

map (new object), *cast* (chgt of type), *index*(Tuple with an indice)

1 to N :

flatMap + a factory method, *handle*

Add elements to a sequence:

startsWith, *endsWith*

Agregate :

collectList, *collectMap*, *count*, *reduce*, *scan*,

Agregate with a boolean :

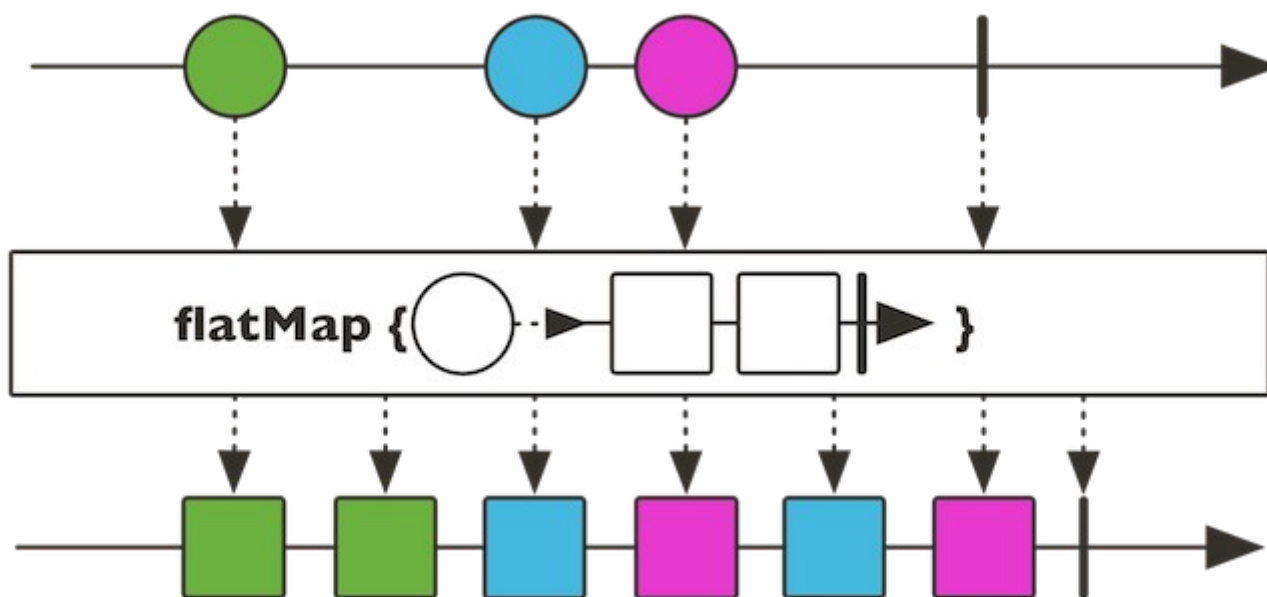
all, *any*, *hasElements*, *hasElement*

Combine multiple streams :

concat, *merge*, *zip*



flatMap





Filters

Filter on arbitrary function :

filter

On the type :

ofType

Ignore all :

ignoreElements

Ignore duplicates :

distinct

Only a subset:

take, takeLast, elementAt

Skip some elements :

skip(Long | Duration), skipWhile



Temporal operators

Associate the event with a timestamp:

elapsed, timestamp

Sequence interrupted if a timeout occurs

between 2 events:

timeout

Sequence at regular intervals:

interval

Add delays

Mono.delay, delayElements, delaySubscription



Reactive Spring

Reactive programming
Spring Reactor
Spring Data Reactive
Spring Webflux



Introduction

Reactive programming also invites itself in
Spring Data

Are supported :

- MongoDB
- Cassandra
- Redis
- ... and jdbc



Reactive access to persistent data

Calls are asynchronous, non-blocking, event-driven

Data is treated as streams

This requires :

- Spring Reactor
- Spring Framework 5
- Spring Data 2.0
- Un reactive driver
- Éventuellement Spring Boot (2.0)



Mixing non-blocking and blocking

If blocking and non-blocking code must be mixed, the main thread executing the event loop should not be blocked.

We can then use the Spring Reactor Scheduler.



Principles

Reactive functionality stays close to Spring Data concepts:

- Reactive Templates API
- Reactive repository
- Returned objects are Flux or Mono



Reactive Template

Template class API becomes :

```
<T> Mono<T> insert(T objectToSave)
<T> Mono<T> insert(Mono<T> object)
<T> Flux<T> insertAll(Collection<? extends T>
    objectsToSave)
<T> Flux<T> find(Query query, Class<T> type
...

```

Example :

```
Flux<Person> insertAll = template
.insertAll(Flux.just(new Person("Walter", "White", 50), //
new Person("Skyler", "White", 45), //
new Person("Saul", "Goodman", 42), //
new Person("Jesse", "Pinkman", 27)).collectList());

```



Reactive Repository

The ***ReactiveCrudRepository<T,ID>*** interface allows for reactive CRUD function implementations.

For example :

```
Mono<Long> count()  
Mono<Void> delete(T entity)  
Flux<T> findAll()  
Mono<S> save(S entity)  
..
```



Request

Like in SpringData, queries can be inferred from function names:

```
public interface ReactivePersonRepository extends
    ReactiveCrudRepository<Person, String> {

    Flux<Person> findByLastname(String lastname);

    @Query("{ 'firstname': ?0, 'lastname': ?1}")
    Mono<Person> findByFirstnameAndLastname(String firstname, String
        lastname);

    // Accept parameter inside a reactive type for deferred execution
    Flux<Person> findByLastname(Mono<String> lastname);

    Mono<Person> findByFirstnameAndLastname(Mono<String> firstname, String
        lastname);
}
```



Dependencies example : *MongoDB with SpringBoot*

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.0.0.BUILD-SNAPSHOT</version>
</parent>
<!-- Brings in particular: spring-data-mongodb and reactor-core -->
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>

    <artifactId>spring-boot-starter-data-mongodb-reactive</artifactId>
  </dependency>
</dependencies>
```




Reactive Spring

Reactive programming
Spring Reactor
Spring Data Reactive
Spring Webflux



Motivation

2 main motivations for Spring Webflux:

- The need for a non-blocking stack allowing to manage concurrency with few threads and to scale with less CPU/memory resources
- Functional programming



Introduction

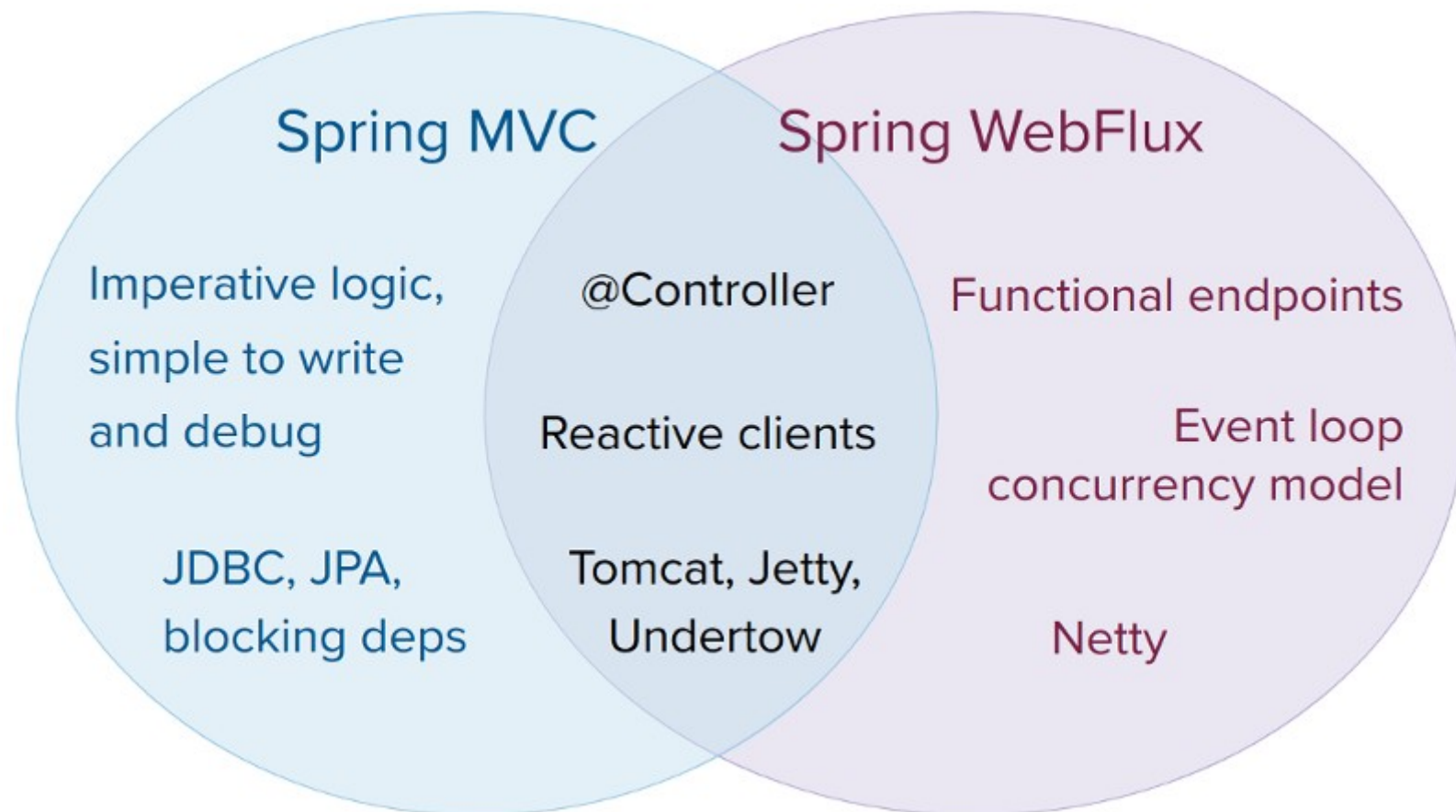
The spring-reactive-web module is the basis for Spring Webflux.

It offers 2 programming models

- Annotated Controllers: Same as Spring MVC with the same annotations.
Controller methods can return reactive types, reactive arguments are associated with *@RequestBody*.
- Functional endpoints: Functional programming based on lambdas.
Ideal for small libs to route and process requests.
In this case, the application is in charge of processing the request from start to finish.



MVC and WebFlux





Servers

Spring WebFlux is supported on

- Tomcat, Jetty, and Servlet 3.1+ containers,
- As well as non-Servlet environments like Netty or Undertow

The same programming model is supported on all these servers

With SpringBoot, the default configuration starts an embedded Netty server



Performance and Scaling

The reactive and non-blocking model does not bring any particular gain in terms of response time.

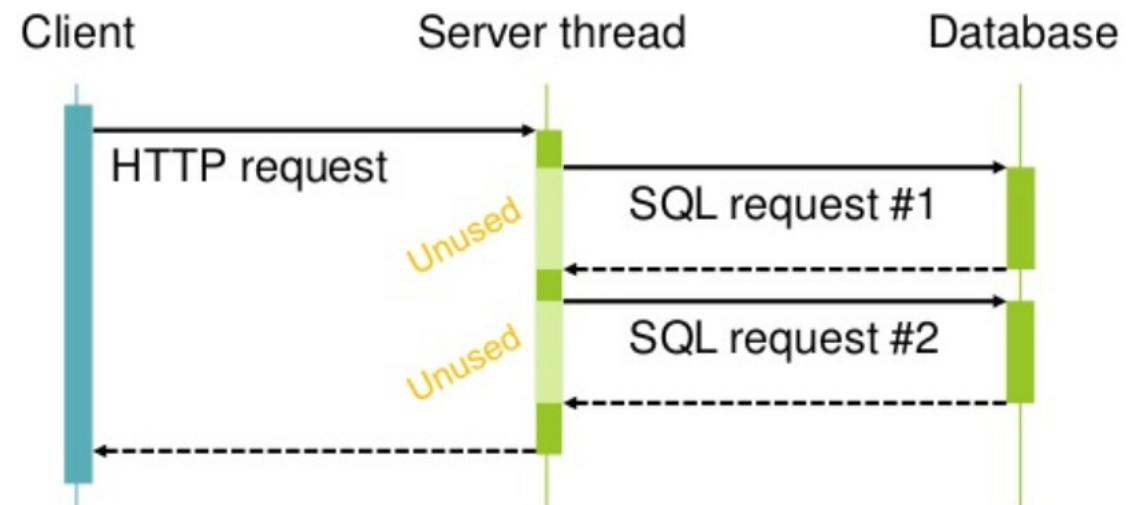
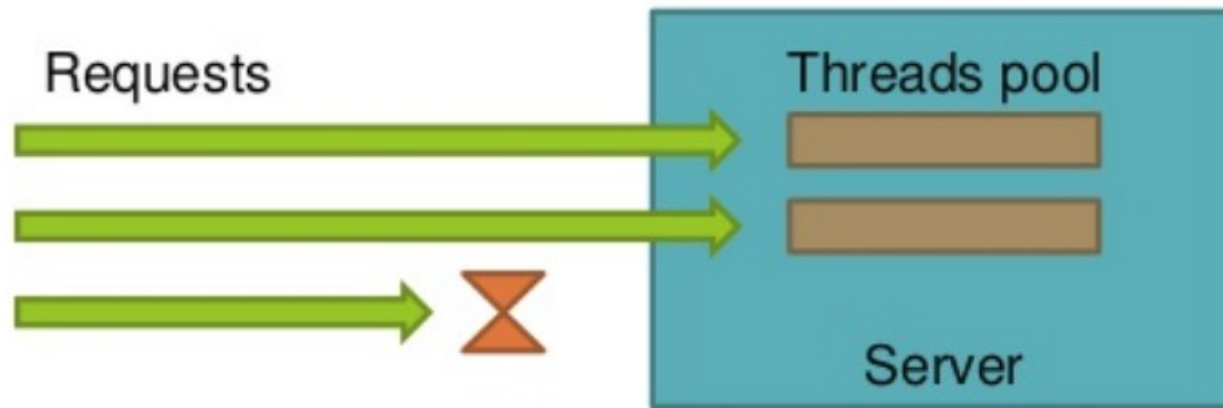
(there is more to do and it may even increase processing time)

The expected benefit is the ability to scale with a small number of fixed threads and less memory.

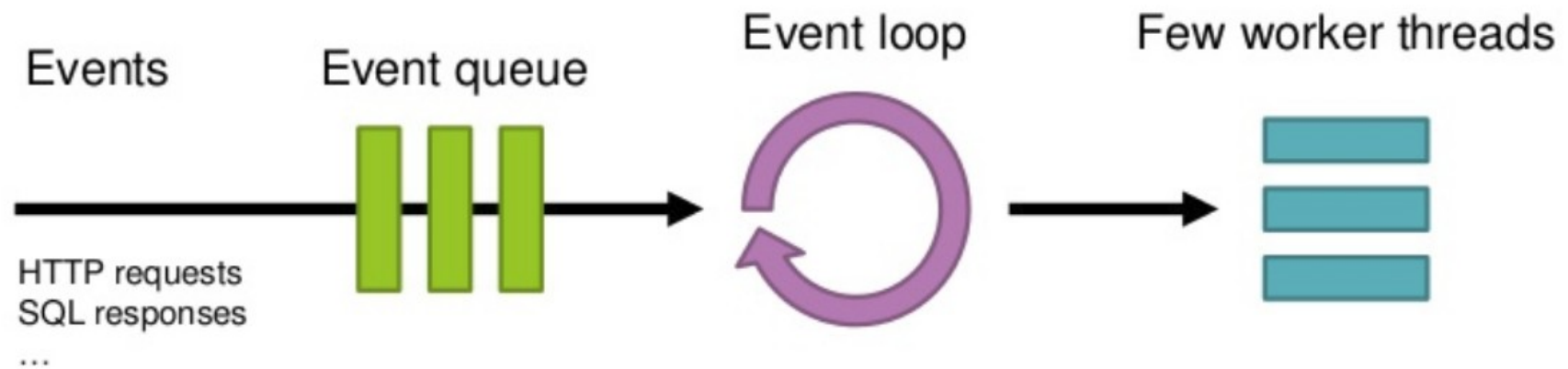
This makes applications more resistant to load.

- To be able to see these benefits, it is necessary to introduce latency, for example by introducing slow or unpredictable IO networks.

Blocking model



Non-blocking model





Threading model

For a fully responsive Spring WebFlux server, one can expect 1 thread for the server and as many threads as CPU for request processing.

If you have to access JPA data for example, it is advisable to use Schedulers which then modifies the number of threads

To configure the server's threading model, one needs to use their specific configuration API or see if Spring Boot offers support.



API

In general the WebFlux API

- accepts as input a Publisher,
- internally adapts it to Reactor types,
- uses it and returns either a Flux or a Mono

In terms of integration:

- Any Publisher can be provided as input
- You have to adapt the output if you want it to be compatible with a library other than Reactor



Annotated controllers

Spring MVC's **@Controller** annotations are therefore supported by WebFlux.

The differences are:

- Core beans like *HandlerMapping* or *HandlerAdapter* are non-blocking and work on reactive classes
- ***ServerHttpRequest*** and ***ServerHttpResponse*** rather than *HttpServletRequest* and *HttpServletResponse*.



Example

```
@RestController
public class PersonController {
    private final PersonRepository repository;
    public PersonController(PersonRepository repository) {
        this.repository = repository;
    }
    @PostMapping("/person")
    Mono<Void> create(@RequestBody Publisher<Person> personStream) {
        return this.repository.save(personStream).then();
    }
    @GetMapping("/person")
    Flux<Person> list() {
        return this.repository.findAll();
    }
    @GetMapping("/person/{id}")
    Mono<Person> findById(@PathVariable String id) {
        return this.repository.findOne(id);
    }
}
```



Controller methods

The methods of the controllers resemble those of Spring MVC (Annotations, arguments and possible return value), with a few exceptions

– Arguments :

- ***ServerWebExchange*** : Encapsulates, request, response, session, attributes
- ***ServerHttpRequest*** and ***ServerHttpResponse***

– Return values :

- ***Flux<ServerSentEvent>***,
Observable<ServerSentEvent> : Data + Metadata
- ***Flux<T>***, ***Observable<T>*** : Only data

– Request Mapping (consume/produce) : *text/event-stream*



Functional *Endpoints*

In this functional programming model, functions (lambda-expression) are used to route and process requests.

The interfaces representing the HTTP interaction (request/response) are immutable

=> Thread-safe needed for reactive model



ServerRequest and ServerResponse

ServerRequest and ***ServerResponse*** are therefore interfaces that provide access via lambda-expression to HTTP messages.

- ***ServerRequest*** exposes the request body as Flux or Mono.
It gives access to HTTP elements (Method, URI, ..) through a separate *ServerRequest.Headers* interface.
`Flux<Person> people = request.bodyToFlux(Person.class);`
- ***ServerResponse*** accepts any Publisher as a body.
It is created via a builder allowing to position the status, the headers and the response body
`ServerResponse.ok()
 .contentType(MediaType.APPLICATION_JSON).body(person);`



HandlerFunction

HTTP requests are handled by a ***HandlerFunction***: a function that takes as input a *ServerRequest* and provides a *Mono<ServerResponse>*

Example :

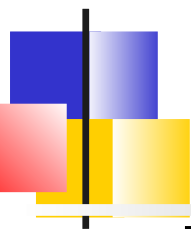
```
HandlerFunction<ServerResponse> helloWorld =  
    request ->  
    ServerResponse.ok().body(fromObject("Hello World"));
```

Typically, similar functions are grouped into a controller class.



Example

```
public class PersonHandler {  
    private final PersonRepository repository;  
  
    public PersonHandler(PersonRepository repository) { this.repository = repository;}  
  
    public Mono<ServerResponse> listPeople(ServerRequest request) {  
        Flux<Person> people = repository.allPeople();  
        return ServerResponse.ok().contentType(APPLICATION_JSON).body(people, Person.class);  
    }  
  
    public Mono<ServerResponse> createPerson(ServerRequest request) {  
        Mono<Person> person = request.bodyToMono(Person.class);  
        return ServerResponse.ok().build(repository.savePerson(person));  
    }  
  
    public Mono<ServerResponse> getPerson(ServerRequest request) {  
        int personId = Integer.valueOf(request.pathVariable("id"));  
        Mono<ServerResponse> notFound = ServerResponse.notFound().build();  
        Mono<Person> personMono = this.repository.getPerson(personId);  
        return personMono  
            .then(person -> ServerResponse.ok().contentType(APPLICATION_JSON).body(fromObject(person)))  
            .otherwiseIfEmpty(notFound);  
    }  
}
```



Mapping with *RouterFunction*

Requests are routed to HandlerFunctions with a ***RouterFunction***:

Takes as input a `ServerRequest` and returns a `Mono<HandlerFunction>`

- Functions are generally not written directly. We use :
RouterFunctions.route(RequestPredicate,HandlerFunction)
allowing to specify the matching rules

Example :

```
RouterFunction<ServerResponse> helloWorldRoute =  
RouterFunctions.route(RequestPredicates.path("/hello-  
world"),  
request -> Response.ok().body(fromObject("Hello World")));
```



Combination

2 routing functions can be composed into a new function via the methods

`RouterFunction.and(RouterFunction)`

`RouterFunction.andRoute(RequestPredicate, HandlerFunction)`

If the first rule does not match, the second is evaluated... and so on



Example

```
PersonRepository repository = ...
```

```
PersonHandler handler = new PersonHandler(repository);
```

```
RouterFunction<ServerResponse> personRoute = RouterFunctions.
```

```
  route(RequestPredicates.GET("/person/{id}"))
```

```
    .and(accept(APPLICATION_JSON)), handler::getPerson)
```

```
  .andRoute(RequestPredicates.GET("/person"))
```

```
    .and(accept(APPLICATION_JSON)), handler::listPeople)
```

```
  .andRoute(RequestPredicates.POST("/person"))
```

```
    .and(contentType(APPLICATION_JSON)), handler::createPerson);
```



Running on a server

To execute a RouterFunction on a server, it must be converted into an `HttpHandler`.

Different techniques are possible but the simplest is to specialize a ***WebFlux*** configuration

- The default configuration brought by ***@EnableWebFlux*** does the needful
- or directly via SpringBoot and the webflux starter



Exemple

@Configuration

@EnableWebFlux

```
public class WebConfig implements WebFluxConfigurer {  
    @Bean  
    public RouterFunction<?> routerFunctionA() { // ... }  
    @Bean  
    public RouterFunction<?> routerFunctionB() { // ... }  
    @Override  
    public void configureHttpMessageCodecs(ServerCodecConfigurer configurator) {  
        // configure message conversion...  
    }  
    @Override  
    public void addCorsMappings(CorsRegistry registry) {  
        // configure CORS...  
    }  
    @Override  
    public void configureViewResolvers(ViewResolverRegistry registry) {  
        // configure view resolution for HTML rendering...  
    }  
}
```



Filter via *HandlerFilterFunction*

Routes controlled by a routing function can be filtered:

```
RouterFunction.filter(HandlerFilterFunction)
```

HandlerFilterFunction is a function taking a `ServerRequest` and a `HandlerFunction` and returning a `ServerResponse`.

The *HandlerFunction* parameter represents the next element in the chain: the handler function or the filter function.



Example :

Basic Security Filter

```
import static org.springframework.http.HttpStatus.UNAUTHORIZED;

SecurityManager securityManager = ...
RouterFunction<ServerResponse> route = ...

RouterFunction<ServerResponse> filteredRoute =
    route.filter((request, next) -> {
        if (securityManager.allowAccessTo(request.path())) {
            return next.handle(request);
        }
        else {
            return ServerResponse.status(UNAUTHORIZED).build();
        }
    });
```




Towards production

Monitoring with actuator
Deployment



Actuator

Spring Boot Actuator provides support for monitoring and managing SpringBoot applications

Monitoring resources can be accessed

- Via HTTP (If Spring MVC)
- Via JMX

Starter :

spring-boot-starter-actuator



Features

The transverse functionalities offered by Actuator relate to:

- App health status
- Getting Metrics
- Security audit
- Traces of HTTP requests
- Viewing the configuration
- ...



Endpoints

Actuator provides many default endpoints:

- **beans** : Beans instantiated by Spring
- **env / configprops** : Configuration properties resolved
- **health** : Health of the app
- **info** : Arbitrary Information. In general, Commit, version
- **metrics** : Metrics
- **mappings** : Mappings configured
- **trace** : Trace of the last HTTP requests
- **docs** : Documentation, Sample Requests and Responses
- **logfile** : Contents of the log file (if exist)

Custom Beans of type **Endpoint**, are automatically exposed



Configuration

Endpoints can be configured by properties.

Each endpoint can be

- On/off
- Secured by Spring Security
- Mapped to another URL

In SB 2.x, only /health and /info endpoints are enabled by default

To enable others:

- *management.endpoints.web.exposure.include=**
- Or list them one by one



Endpoint */health*

The information provided is used to determine the status of an application in production.

- The probe can be used by monitoring tools responsible for alerting when the system goes down (Kubernetes for example)

By default, the endpoint displays a global status but Spring can be configured to have each subsystem (beans of type `HealthIndicator`) display its status:

```
management.endpoint.health.show-details= always
```



Indicators provided

Spring provides the following health indicators when appropriate:

- ***CassandraHealthIndicator*** : Cassandra is up.
- ***DiskSpaceHealthIndicator*** : Enough free space ?
- ***DataSourceHealthIndicator*** : Connection to a datasource
- ***ElasticsearchHealthIndicator*** : Elasticsearch is up.
- ***JmsHealthIndicator*** : JMS broker is up.
- ***MailHealthIndicator*** : Mail server is up.
- ***MongoHealthIndicator*** : Mongo is up.
- ***RabbitHealthIndicator*** : Rabbit is up
- ***RedisHealthIndicator*** : Redis is up.
- ***SolrHealthIndicator*** : Solr is up
- ...



Application Information

By default , the ***/info*** endpoint does not display anything.

If you want the details on Git:

```
<dependency>
  <groupId>pl.project13.maven</groupId>
  <artifactId>git-commit-id-plugin</artifactId>
</dependency>
```

If you want build information:

```
<plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
  <executions>
    <execution>
      <goals>
        <goal>build-info</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```




Metrics

The metrics endpoint provides access to all kinds of metrics. We find :

- System: Memory, Heap, Threads, GC
- Data Source: Active Connections, Pool Status
- Cache: Size, Hit and Miss Ratios
- Tomcat Sessions



Endpoints with SpringBoot 2

/auditevents : List of security events (login/logout)

/conditions : Replaces /autoconfig, report on auto-configuration

/configprops – Beans annotated by *@ConfigurationProperties*

/flyway ; BD Flyway Migration Information

/liquibase : Liquibase migration

/loggers : Allows you to view and modify the log level

/scheduledtasks : Scheduled tasks

/sessions : HTTP sessions

/threaddump : Thread dumps



Towards production

Monitoring with actuator
Deployment



Introduction

Several alternatives to deploy a Spring-boot application:

- Stand-alone application
- Archive war to deploy on application server
- Linux or Windows service
- Docker image
- The cloud



Application stand-alone

The Spring-boot Maven plugin is used to generate the stand-alone application:

```
mvn package
```

Creates an executable archive containing application classes and dependencies in the *target* directory

To run :

```
java -jar target/artifactId-version.jar
```



Manifest file

Manifest-Version: 1.0

Implementation-Title: documentService

Implementation-Version: 0.0.1-SNAPSHOT

Archiver-Version: Plexus Archiver

Built-By: dthibau

Start-Class: org.formation.microservice.documentService.DocumentsServer

Implementation-Vendor-Id: org.formation.microservice

Spring-Boot-Version: 1.3.5.RELEASE

Created-By: Apache Maven 3.3.9

Build-Jdk: 1.8.0_121

Implementation-Vendor: Pivotal Software, Inc.

Main-Class: org.springframework.boot.loader.JarLauncher



Création de war

To create a war, it is necessary to:

- Provide a subclass of ***SpringBootServletInitializer*** and override the `configure()` method. This allows to configure the application (Spring Beans) when the war is installed by the servlet container.
- To change the packaging element of the pom.xml to war
`<packaging>war</packaging>`
- Then exclude tomcat libraries
For example by specifying that the dependency on the Tomcat starter is provided

```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-tomcat</artifactId>  
  <scope>provided</scope>  
</dependency>
```



Example

```
@SpringBootApplication
public class Application extends SpringBootServletInitializer {
    @Override
    protected SpringApplicationBuilder configure(
        SpringApplicationBuilder application) {
        return application.sources(Application.class);
    }
    public static void main(String[] args) throws Exception {
        SpringApplication.run(Application.class, args);
    }
}
```




Linux service

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <configuration>
        <executable>true</executable>
      </configuration>
    </plugin>
  </plugins>
</build>
```

=> target/artifactId.jar is executable !

=> ln -s target/artifactId.jar /etc/init.d/artifact
service artifact start



Cloud

Spring Boot executable jars are ready to deploy on most PaaS platforms

The reference documentation offers support for:

- Cloud Foundry
- Heroku
- OpenShift
- Amazon Web Services
- Google App Engine



Example CloudFoundry/Heroku

Cloud Foundry

```
cf login
```

```
cf push acloudyspringtime -p target/demo-0.0.1-SNAPSHOT.jar
```

Heroku

Mise à jour d'un fichier Procfile :

```
web: java -Dserver.port=$PORT -jar target/demo-0.0.1-SNAPSHOT.jar
```

```
git push heroku master
```



Annex



Format YAML

YAML (Yet Another Markup Language) is an extension of JSON, it is very convenient and very compact to specify hierarchical configuration data.

... but also very sensitive, to indentation for example



Exemple *.yml*

```
environments:  
  dev:  
    url: http://dev.bar.com  
    name: Developer Setup  
  prod:  
    url: http://foo.bar.com  
    name: My Cool App
```

Produce :

```
environments.dev.url=http://dev.bar.com  
environments.dev.name=Developer Setup  
environments.prod.url=http://foo.bar.com  
environments.prod.name=My Cool App
```



Arrays

YAML arrays are represented by properties with an index.

```
my:
  servers:
    - dev.bar.com
    - foo.bar.com
```

become:

```
my.servers[0]=dev.bar.com
my.servers[1]=foo.bar.com
```



RESTFul principles



Auto-description

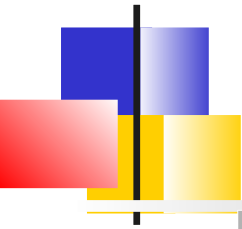
A well-designed API is understandable by a developer without having to read the documentation

- the API is self-describing.
- An API materializes directly in the URL of HTTP requests sent to the server exposing the resource.

Example of an HTTP request on a resource from the UBER company API :

GET https://api.uber.com/<version>/partners/payments

=>A developer intuitively knows that he will find in this resource the information regarding the payments received by the partner UBER (independent driver).



Principles

One ressource $\leq = >$ One entity

URI	GET	PUT	POST	DELETE
Collections : <i>http://api.example.com/produits</i>	List all products	Replace the list with another list	Insert a new element un the collection	Remove the collection
Element, <i>http://api.example.com/produits/5</i>	Retreive product with ID 5	Replace the l'élément	Treat the element as a collection and add an entry to it	Remove the element



Naming conventions

For objects associated to the main entity. It is recommended to use a hierarchical structure : /objets/{objet_id}/sous_objets

GET /calendar/meetings/{meeting_id}/meeting_room

The resource operation can be specified if the HTTP method is not sufficient:

GET /calendar/meetings/{meeting_id}/attendees/search

Request parameters

I

Paramètre	Utilisation	Exemple
Path Parameter	Pour l'identification seulement : <ul style="list-style-type: none">▪ Uniquement un ID, toujours suivant l'entité à laquelle il se réfère▪ Paramètre obligatoire	{id} = 123 http://.../accounts/123/transactions
Query Parameter	Pour la gestion de résultat - filtrer, trier, ordonner, grouper les résultats (paramètres courts) : <ul style="list-style-type: none">▪ Paramètres techniques optionnels▪ Valeurs sont définies et documentées dans la spécification de l'API	http://.../transactions ? from = NOW & sort = date:desc & limit = 50
Header	Pour la gestion du contexte d'application et de la sécurité <ul style="list-style-type: none">▪ Utilisé par les navigateurs, les applications clientes et autres pour transmettre des informations sur le contexte de la demande▪ Utilisé pour transmettre les paramètres d'authentification <p><u>NB</u> : Ne pas utiliser pour transmettre les paramètres fonctionnels</p>	Authorization : Bearer XXXXXXXX
Body	Pour les données fonctionnelles <ul style="list-style-type: none">▪ Utilisé pour transmettre des informations fonctionnelles▪ Doit être un objet JSON	{ "name": "phone", "category": "tech", "max_price": 45 }



Return status

HTTP return codes are used to determine the result of a request or to indicate an error to the client.

They are standard

- **1xx** : Information
- **2xx** : Success
- **3xx** : Redirection
- **4xx** : Error client
- **5xx** : Error server



Success

"2XX" return codes are the results of successfully executed queries. The most common code is code 200. There are others that respond to more specific cases.

- **200 - OK** : Any successful request
- **201 - Created & Location** : Creation of a new object. The link or the identifier of the new resource is sent in the response
- **204 - No content** : Update or delete an object (with an empty response)
- **206 - Partial Content** : A paginated list of objects for example



Client errors

"4XX" return codes indicate that the request sent by the client cannot be executed by the server.

- **400 - Bad Request** : The request is incorrect. Generally a bad conversion
- **401 - Unauthorized** : The request requires authentication
- **403 - Forbidden** : Resources not accessible for the authenticated user
- **404 - Not Found** : The requested object does not exist
- **405 - Method Not Allowed** : The URL is good but the HTTP method is not
- **406 - Not Acceptable** : Requested headers cannot be satisfied. (Accept-Charset, Accept-Language)
- **409 - Conflict** : For example: Attempting to create a new user with an already existing email address
- **429 - Too Many Requests** : The client made too many requests in a given time frame



Server errors

"5XX" return codes indicate that the server encountered an error. The most common types of server errors are:

- **501 - Not Implemented** : The method (GET, PUT, ...) is not known to the server for any resource
- **502 - Bad Gateway ou Proxy Error** : The response from the backend is not understood by the API Gateway
- **503 - Service Unavailable** : API out of service, under maintenance, etc.
- **504 - Gateway Time-out** : Timeout expired