Spring and Spring Boot

David THIBAU – 2022

david.thibau@gmail.com

Agenda

- **Introduction**
  - History
  - IoC and Dependency Injection
- **Annotations**
  - Configuration classes
  - @Component and stereotypes
  - Dependency injection
  - Environment and profiles
- **Spring Reactive**
  - Introduction
- **Spring Boot**
  - Auto-configuration
  - Starters
  - Configuration SpringBoot
- **Spring and persistence**
  - Spring Data
  - Spring Data JPA
- **RestFul APIs**
  - Spring MVC for RestFul APIs
  - Spring Boot for Restful APIs
  - Jackson and serialization
  - Exceptions, CORS, OpenAPI
- **Services interactions**
  - Rest Client
  - Messaging
- **Spring Security**
  - Principles, stateful/stateless models
  - Autoconfiguration Spring Boot
  - OpenIdConnect and oAuth2
- **Spring and tests**
  - Spring Test, @SpringBootTest
  - Auto-configured tests
- **Toward production**
  - Actuator
  - Deployment
- **Annexes**
  - Auto-configuration mechanism
  - NoSQL
  - Spring MVC Legacy
  - Spring Reactive

Introduction

History

IoC and Dependency Injection

Spring and Dependency Injection

History

- Spring is an ***OpenSource*** project supported by a big commercial company : ***Pivotal Software***
- The project started with Rod Johnson and Jurgen Holler in 2002.
  It was an alternative to the J2EE specification supported by Sun then Oracle.
- Nowadays, this the most used Java Framework !!

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, each responding to a specific problemtatic .

All theses project have common features :

- They lay on same foundation (Spring Core)
- It allows to write clean, modular and testable code
- Avoid to code technical aspects (plumbing)
- Be portable : All you need is … a JVM

Some projets

Spring core : Basic foundation. Rely on the IoC pattern, low-level services

Spring Security : All about securing a Java (web) application

Spring Data : Common approach to access to persistent data (SQL, NOSQL)

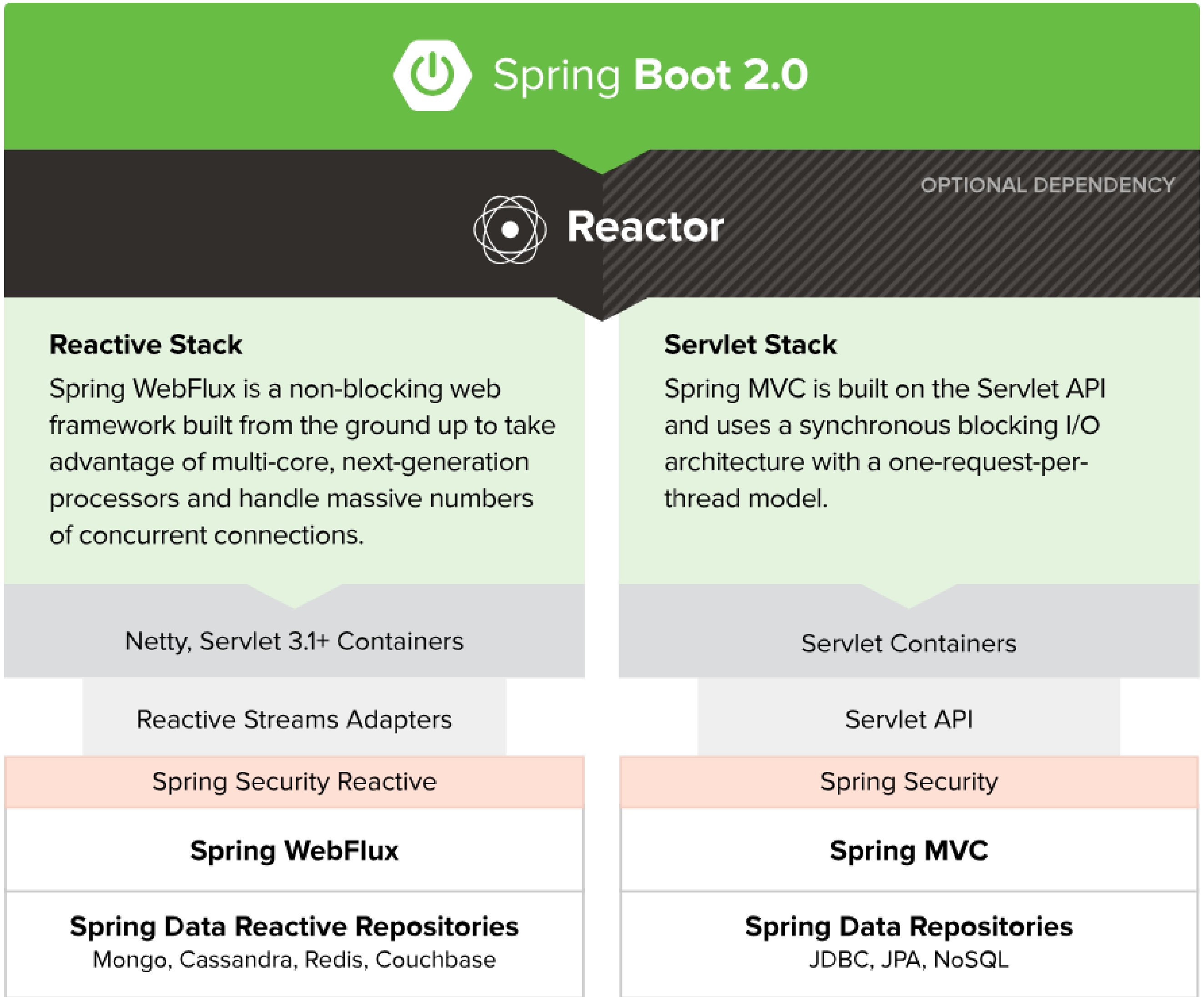
Spring Integration : How to integrate legacy application together

Spring Batch : Optimizing batch jobs

Spring Cloud : Deploy micro-services in the cloud

…

Web Stacks


Spring Boot 2.0
Reactor
OPTIONAL DEPENDENCY
Reactive Stack
Spring WebFlux is a non-blocking web framework built from the ground up to take advantage of multi-core, next-generation processors and handle massive numbers of concurrent connections.
Servlet Stack
Spring MVC is built on the Servlet API and uses a synchronous blocking I/O architecture with a one-request-per-thread model.
Netty, Servlet 3.1+ Containers
Servlet Containers
Reactive Streams Adapters
Servlet API
Spring Security Reactive
Spring Security
Spring WebFlux
Spring MVC
Spring Data Reactive Repositories
Mongo, Cassandra, Redis, Couchbase
Spring Data Repositories
JDBC, JPA, NoSQL

Introduction

History

IoC and Dependency Injection

Spring and Dependency Injection

IoC Pattern

Inversion Of Control

- The problem :

How to make the controller-based web architecture work with the database when these are developed by different teams?

- The OO paradigm response *:*

Use interface !

Illustration

- In a web controller, we want to provide a method which list all the movies of a specific director.
  This method can rely on a DAO (Data Access Object) layer, which allow to retreive all the films from a PersistentStore
- Interface :

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

- Controller

```
public class MovieController...
   public List<Movie> moviesDirectedBy(String arg) {
        List<Movie> allMovies = finder.findAll();

        return allMovies.stream()
               .filter(m -> !m.getDirector().equals(arg))
               .collect(Collectors.toList()) ;

   }
```

Implementation ?

- Even if the code is decoupled, somewhere you have to insert a concrete class that implements the finder interface.
  - For example, in the constructor of the *MovieLister* class.

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

- Arggh, we have lost our independance !!

IoC and framework

The **IoC (Inversion Of Control) pattern** means that it is no longer the developer's code which has control of execution but the framework

The framework is then responsible for instantiating the objects, calling the methods, releasing the objects, etc.

Developer code is reduced to business code.

IoC versus Dependency Injection

- **Dependency injection** is a specialization of the IoC pattern

- The framework is responsible for initializing object attributes.

- In the previous illustration, it initializes the interface variable with an implementation object.

How to inject

- There are 3 ways to inject a dependency :
  - **By constructor :**
    The recommanded way and the most used in Spring
  - **With a setter method**
  - **With an interface or annotation**
  - **Direct access to the attribute**
  -

Constructor injection

- La classe MovieController doit déclarer un constructeur permettant d'initialiser l'attribut MovieFinder.

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

- L'objet *finder* se fait injecter une propriété via le constructeur : le nom du fichier .

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

Container configuration

- Configuring the framework consists of specifying the classes it must manage (beans) and how it injects dependencies.
- This is done:
  - Via XML configuration files. (Legacy)
  - Via Java configuration classes and Java annotations (Since Java5)
  - By applying default configurations (SpringBoot)

XML Configuration sample

```
<beans>
   <bean id="movieController" class="spring.MovieController">
        <property name="finder" ref="colonMovieFinder"/>
   </bean>
<bean id="colonMovieFinder" class="spring.ColonMovieFinder">
        <property name="filename" value="movies1.txt"/>
   </bean>
</beans>
```

BeanDefinition

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

- The qualified name of the class of the bean
- The behaviour configuration (scope, call-back methods, ...).
- References to other beans (dependencies)
- ...

Naming

- Each bean has one (or several) identifier : its **name**.
- The convention is to use standard naming Java convention for an instance attribute.
- Other names are called *aliases*.

ApplicationContext

- When Spring boots from the configuration, it creates an object implementing the ***ApplicationContext*** interface

- This provides some methods

`java.lang.Object getBean(java.lang.String name)` : Return an instance, which may be shared or independent, of the specified bean.

`<T> T   getBean(java.lang.Class<T> requiredType)` : Return the bean instance that uniquely matches the given object type, if any.

`java.lang.Class<?> getType(java.lang.String name)` : Determine the type of the bean with the given name.

`boolean isSingleton(java.lang.String name)` : Is this bean a shared singleton ?

…

Test

```
public void testWithSpring()
   throws Exception {


  // Instantiate all the beans of the configuration
   ApplicationContext ctx =
                 new FileSystemXmlApplicationContext("spring.xml");

   MovieLister lister = (MovieLister)ctx.getBean("MovieLister");

   Movie[] movies = lister.moviesDirectedBy("Sergio Leone");

   assertEquals("Once Upon a Time in the West",movies[0].getTitle());

}
```

Aspects (AOP)

During configuration, it is also possible to add **technical/transversal services** to the objects managed by the framework:

- Security: *@RolesAllowed*
- Transaction: *@Transactional*
- ...

Examples

With an IoC framework like Spring, a developer can:

- Write a method that executes in a database transaction without using the transaction API
- Make a method accessible remotely without using a remote API
- Define a method protected by ACLs without using security API code
- Define a message handler method without using the broker API

Annotations

Configuration classes

@Component and stereotypes

Dependency injection

Configuration properties

Environment and profiles

Alternative to XML

- Instead of XML, it is possible to use Java annotations and configuration classes to define beans to be instantiated.
- Each approach has its pros and cons.
- They can also be combined.

- In fact, configuration via Java is much more practical and has therefore become essential.

Configuration classes

Since Spring 3.0, configuration can be made with Java classes annotated with ***@Configuration***

- These classes are mainly constituted of methods annotated with ***@Bean*** which defines the technical beans instantiated and managed by Spring
- The purpose of these classes are to centralized all the beans related to a specific area of the application (ex : security, database access, ...)

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

Configuration composition

The ***@Import*** annotation all to import another Configuration Class

```
@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);
```

Reading configuration

2 ways for the framework to process the annotations:

- Indicate the location of the configuration classes
- Specify a package to scan to discover the annotations

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();
}
```

Package scan:

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

@Bean attributes

@Bean has 3 attributes :

- ***name*** : The name and the aliases of the bean
- ***init-method*** : Call-back method called after initialization
- ***destroy-method*** : Call-back method before deletion

```
@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();
}
}
```

To define the scope of the bean, you must use the ***@Scope*** annotation

@Enable annotations

@Configuration classes are generally used to configure external resources (a datasource, a Kafka client, etc.)

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

@Enable Examples

- ***@EnableWebMvc*** : Enable Spring MVC in an application
- ***@EnableCaching*** : Allows the use of annotations *@Cachable*, ...
- ***@EnableScheduling*** : Allows the use of annotations *@Scheduled*
- ***@EnableJpaRepositories*** : Allows you to scan classes Repository
- ...

Annotations

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

Introduction

@Configuration are often used to configure external elements like a database, a smtp server, ...

To declare applicative beans, we generally used the ***@Component*** and its derivatives

- This annotation is set on a class and must be scanned by Spring

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 bean classes corresponding thanks to the ***@ComponentScan*** annotation.

- This annotation can specify a package. (The default one is the current package)
- It can be set on a configuration class or the root class of the application

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

Stereotypes

@Component is a generic stereotype .

Since Spring 2.5, other stereotypes are provided : ***@Repository***, ***@Service, @Controller*** and ***@RestController*** are some specialisation of *@Component*.

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

For example: JSON serialization for @RestController

Life cycle

Beans, whether instantiated via Configuration classes or stereotyped annotations, can have 3 life cycles (or scope):

- **Singleton**: There is only one instance of the object (which is therefore shared). Ideal for “stateless” beans.
  => This is the overwhelming majority of cases.
- **Prototype**: Each time the bean is used via its name, a new instance is created.
  => Almost Never
- **Custom object “scopes”**: Their life cycle is generally synchronized with other objects, such as an HTTP request, an http session, a BD transaction
  => Some beans provided by Spring. Ex: EntityManager

@Scope

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

```
@Scope("prototype")
@Repository
public class MovieFinderImpl implements
  MovieFinder {
// ...
}
```

Méthodes de call-back

Spring supports call-back annotations ***@PostConstruct*** and ***@PreDestroy***

```
@Component
public class CachingMovieLister {
  @PostConstruct
  public void populateMovieCache() {
    // Initialisation après construction...
  }
  @PreDestroy
  public void clearMovieCache() {
  // Nettoyage avant destruction...
  }
}
```

Annotations

Configuration classes

@Component and stereotypes

Dependency injection

Configuration properties

Environment and profiles

@Autowired

The ***@Autowired*** annotation is set on setter methods, arbitrary methods, constructors or attributes

It asks Spring to inject a bean of the argument type

- Generally only one bean is a candidate for injection

@Autowired has a *required* additional attribute, (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;
// ...
}
```

Implicit injection

- In the latest versions of Spring, the *@Autowired* annotation is not necessary when using constructor injection :
  - Generally the attribute is declared as final

- This is called implicit injection, it is an injection by type

Implicit injection

```
@Controller
public class MovieLister {
  private final MovieFinder finder ;


  public MovieLister(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;
   }
```

Exceptions due to auto-wiring

- @Autowired can cause Spring startup exceptions.
  - Cas 1 : Spring cannot find Bean definitions matching type:

```
UnsatisfiedDependencyException,
  No qualifying bean of type '' available:
  expected at least 1 bean which qualifies as autowire candidate.
```

  - Cas 2 : Spring finds multiple Beans of the requested type

```
UnsatisfiedDependencyException,
  No qualifying bean of type '' available:
  expected single matching bean but found 2.
```

@Resource, @Qualifier

@Resource and ***@Qualifier*** allows to use the name of the bean for the injection.

- These annotations specify the ***name*** of the bean
- If the *name* is not specified, the name of the bean to inject corresponds to the name of the property

Example

```
public class MovieRecommender {

  @Resource(name="myPreferenceDao")
  private CustomerPreferenceDao cpDao;

  // Le nom du bean recherché est "context"
  @Resource
  private ApplicationContext context;

  // Plusieurs beans implémentent Formatter
  @Autowired
  @Qualifier("fooFormatter")
  private Formatter formatter;

  public MovieRecommender() {
  }
// ...
}
```

Annotations

Configuration classes

@Component and stereotypes

Dependency injection

Configuration properties

Environment and profiles

Introduction

Spring also allows injecting simple values (String, integers, etc.) into bean properties via annotations :

- ***@PropertySource*** allows to specify a *.properties* file allowing to load configuration values (keys/values)
- ***@Value*** allows bean properties to be initialized with an SpEl expression referencing a configuration key

This requires the presence of a bean ***PropertySourcesPlaceholderConfigurer***[1]

1. Automatically available with SpringBoot

Exemple

```java
@Configuration
 @PropertySource("classpath:/com/myco/app.properties")
 public class AppConfig {

     @Value("${my.property:0}")  // The file app.properties defines the value for the key "my.property"
     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"));  // app.properties defines "testbean.name"
         return testBean;
     }
 }
```

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 file, command line argument or other...
  - **profiles** : Named groups of Beans, beans are registered only if the profile is activated at startup

@Profile

- Any *@Component* or *@Configuration* can be annotated with ***@Profile*** to limit its loading

```
@Configuration
@Profile("production")
public class ProductionConfiguration {

    // ...

}

@Service
@Profile("kafka")
public class MyKafkaServiceImpl implements MyService {

    // ...

}
```

Example

Dev and prod databases

```
@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");
    }
}
```

Activating a profile

Programmatically:

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

Command line :

System property

```
java -jar myJar -Dspring.profiles.active="profile1,profile2"
```

SpringBoot argument

```
java -jar myJar --spring.profiles.active="profile1"
```

Spring Reactif

Introduction

Pattern et *ReactiveX*

Reactive programming:

- is based on the ***Observable*** pattern which is a combination of the *Observer* and *Iterator* patterns
- uses functional programming to easily define operators on the stream of events
- is formalized by the ReactiveX API. Many implementations exist (RxJS, RxJava, Rx.NET)

Reactive Streams

Reactive Streams completes *ReactiveX*

The specification aims to define a standard for the 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
- It does not define operators (each implementation has its own operators)
- An implementation exists in Java9: *Flow API*

Interfaces Reactive Streams

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

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

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

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

Reactor

Reactor focuses on server-side reactive programming.

- It is developed jointly with Spring.
- It mainly provides ***Mono*** and ***Flux*** types representing sequences of events
- It offers a set of operators aligned with *ReactiveX*.
- It is an implementation of *Reactive Streams*

Spring Reactive Ecosystem

- Reactor is the building block of Spring's reactive stack.
- It is used in all responsive projects:
  - ***Spring WebFlux*** : reactive backend
  - ***Spring Data*** *: MongoDB, Cassandra, R2DBC*
  - ***Spring Reactive Security*** *: Reactive security on top of Webflux*
  - ***Spring Test*** *: io.reactor.test :*
  - ***Spring Cloud*** *: Spring Cloud Data Stream, Spring Gateway, ...*

Mono and Flux

Reactor brings mainly 2 types :

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

These are implementations of ***Publisher*** which defines one method :
`void subscribe(Subscriber<? super T> s)`

The stream starts transmitting only if there is a subscriber

Flux

A ***Flow<T>*** represents an asynchronous sequence of 0 to N events, optionally terminated by an *end* signal or an *error*.

The events are translated by method calls on the subscribers:

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

Mono

- ***Mono<T>*** represents a sequence of 0 to 1 event, optionally terminated by an end signal or an error
- Mono offers a subset of Flux operators

Exemple Reactor

```
userService.getFavorites(userId)
           .flatMap(favoriteService::getDetails)
           .switchIfEmpty(suggestionService.getSuggestions())
           .take(5)
           .subscribe(uiList::show, UiUtils::errorPopup);
```

SpringBoot

Auto-configuration
Starters SpringBoot
Development SpringBoot
Configuration properties

Introduction

Spring Boot was designed to simplify the startup and development of new Spring applications

- Easy use of existing techologies (Web, DB, Cloud)
- Can start with « No beans configuration »

Auto-configuration

- The main concept of SpringBoot is **auto-configuration**
- SpringBoot is able to automatically detect the nature of the application and configure the required beans
  - This allows you to start quickly and gradually override the default configuration for application needs
- The mechanisms are different depending on the language: Groovy, Java or Kotlin

Auto-configuration (Java)

Depending on the libraries present at run time, Spring Boot creates all the necessary technical beans with a default configuration.

- For example, if it notices that Web libraries are present, it starts an embedded Tomcat server on port 8080 and applies the default configuration of Spring MVC or Spring Webflux
- If it notices that the H2 driver is in the classpath, it automatically creates a pool of connections to the database
- Etc…

Customizing the configuration

The default configuration can be overridden by different means

- 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 based on profiles
- With configuration classes **redefining default beans**
- By using **specific classes** of the framework (example *Configurer classes)

Autres apports de SpringBoot

Apart from self-configuration, SpringBoot offers other benefits for developers:

- Simplifying the management of dependencies towards OpenSource libraries via ***starters***
- Project creation wizard with ***SpringInitializer***
- ***Central point*** of configuring properties with *application.properties/yml*
- Plugins for **IDEs** (SpringTools suite or other)
- Plugins for **build tools** (Gradle and Maven)

Gestion des dépendances

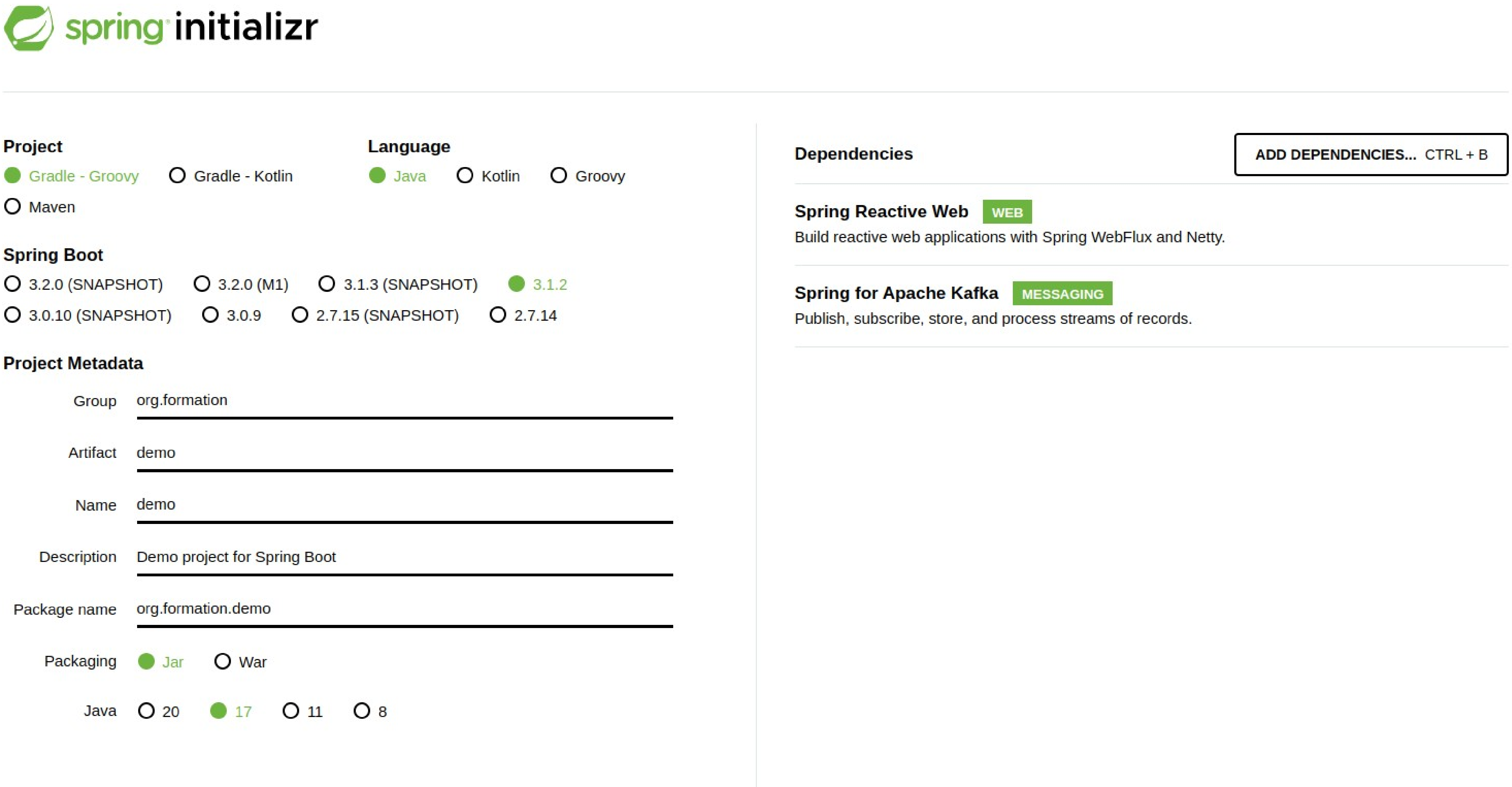
Dependency management is simplified thanks to **starters**.

These are groups of dependencies allowing a technology to be integrated or providing a functionality which are declared in the build file.

=> For developers, this mechanism extremely simplifies dependency versioning.

Only one version number left to manage: That of SpringBoot

https://start.spring.io/



Files generated by the wizard

- *.gitignore, Help.md*
- Build script (*mvnw* ou *gradlew*)
- Starter class (*src/main/java*)
- Configuration test class (*src/test/java*)
- Configuration file for properties (*src/main/resources*)

Example Gradle

```
plugins {
  id 'java'
  id 'org.springframework.boot' version '3.1.2'
  id 'io.spring.dependency-management' version '1.1.2'
}

group = 'org.formation'
version = '0.0.1-SNAPSHOT'

java {
  sourceCompatibility = '17'
}

repositories { mavenCentral() }

dependencies {
  implementation 'org.springframework.boot:spring-boot-starter-webflux'
  implementation 'org.springframework.kafka:spring-kafka'
  testImplementation 'org.springframework.boot:spring-boot-starter-test'
  testImplementation 'io.projectreactor:reactor-test'
  testImplementation 'org.springframework.kafka:spring-kafka-test'
}

tasks.named('test') {   useJUnitPlatform()
```

Maven/Gradle Plug-ins

The initializer creates scripts (mvnw or gradlew) for Linux and Windows environments.

- These are build tool wrappers ensuring that all developers use the same version of the build tool.

The most important command in a Maven context:

./mvnw clean package

=> Generate a *fat-jar*

The application can then be started from the command line by:

java -jar target/myAppli.jar

With gradle :

```
gradle build
java -jar build/libs/mymodule-0.0.1-SNAPSHOT.jar
```

Other tasks

Package an OCI image :
  spring-boot:build-image

Run the application
  spring-boot:run

Run with test classpath
  spring-boot:***test-run***

Build a native image
  package -***Pnative***

Start integration tests
  spring-boot:***start***
  spring-boot:***stop***

Fill the */actuator/info* endpoint with build informations
  spring-boot:***build-info***

Starter class

The startup class is annotated via ***@SpringBootApplication***, which includes:

- ***@Configuration***: Allows you to define @Bean methods
- ***@ComponentScan***: Scanning annotations in sub-packages
- ***@EnableAutoConfiguration***: Enable SpringBoot auto-configuration mechanism

```
@SpringBootApplication
public class DemoApplication {

public static void main(String[] args) {
  // Création du contexte Spring
  SpringApplication.run(DemoApplication.class, args);
}


}
```

Test class

- The test class is annotated via ***@SpringBootTest*** :
  - Allows you to create the Spring context before running the junit5 test

```
@SpringBootTest
class DemoApplicationTests {

@Test
void contextLoads() {
  / If the test passes, the SpringBoot configuration is OK
}


}
```

SpringBoot

Auto-configuration

Starters SpringBoot

Development SpringBoot

Configuration properties

Most important starters

<u>Web</u>

-web : Web Application, RESTFul API

-reactive-web : Idem with reactive programming

<u>Core :</u>

-logging : Logging with logback (Always present)

-test : Junit, AssertJ, Hamcrest et Mockito (Always present)

Development Starters

-devtools : Features for development

***-lombok :** Simplifying Java code

***-configuration-processor :** Completing application configuration properties available in the IDE

***-docker-compose**[1] : Support for starting support services via docker-compose (BD, Kafka, etc.)

***-graalvm-native**[1] : Support for building native images

***-modulith**[1] : Support for building modular monolith applications

1. Since Spring 3.x

Security

- ***-security*** : *Spring Security*, protect URLs and business method
- ***-oauth2-client*** : Obtaining tokens with OpenID Connect
- ***-oauth2-resource-serve**r : Protect URLs via *oAuth*
- ***-ldap*** : LDAP integration
- ***-okta*** : Integration with Okta

Starters SQL

jdbc : JDBC with a connexion pool

Spring Data JPA: Spring Data with Hibernate and JPA

Spring Data JDBC : Spring Data with jdbc

Spring Data R2DBC : Spring Data with reactive *jdbc*

MyBatis : Framework MyBatis

LiquiBase Migration : Schema migration with Liquibase

Flyway Migration : Schema migration with Flyway

JOOQ Access Layer : Fluent API to build SQL request

*-***<drivers>*** : Driver JDBC (MySQL, Postgres, H2, HyperSonic, DB2)

Starters NOSQL

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

***-data-neo4j** : Graph oriented store with Neo4j

***-data-couchbase *-data-reactive-couchbase** : Base NoSQL CouchBase

***-data-redis *-data-reactive-redis** : Base NoSQL Redis

***-data-geode** : Storing data with Geode

***-data-elasticsearch** : Access to ElasticSearch

***-data-mongodb *-data-reactive-mongodb** : Base NoSQL MongoDB

Messaging

-integration: Spring Integration (High-level abstraction to implement integration patterns in a declarative way)

-kafka: Integration with Apache Kafka

-kafka-stream: Integration with KafkaStream API

-rabbitmq: Integration with Rabbit MQ

-activemq5 : ActiveMQ with JMS

-artemis : ApacheMQ with Artemis

*-***pulsar, *-reactive-pulsar*** : Messaging PULSAR

-websocket : Servlet with STOMP and SockJS

-rsocket : SpringMessaging and Netty

-camel : Integration with Apache Camel

-solacePubSub : Integration with Solace

Other Starters Web

<u>HTML templates engine</u>

***-thymeleaf :** *Spring MVC* with *ThymeLeaf view*

***-mustache** : *Spring MVC* with *Mustache*

***-groovy-templates** : *Spring MVC with Groovy templates*

***-freemarker**: *Spring MVC* with *freemarker*

<u>Other</u>

***-graphql** : *API GraphQL*

-rest-repository, restrepository-explorer, *-hateoas : API RestFul generation from SpringData repositories

***-jersey** : Restful API Restful with JAX-RS and Jersey

***-webservices**: SOAP services

-vaadin, *-hila : Framework for web applications

Other Starters

I/O

***-batch** : Batchs jobs

***-mail** : Sending and receiving emails

***-cache**: Support for implementing caches

***-quartz** : Support for Scheduling

***-shell** : Support for CLI

Ops

***-actuator** : Monitoring points via REST or JMX

***-spring-boot-admin (client et serveur)** : UI from Actuator

*-***sentry*** : Integration sentry (monitoring performance)

Spring Cloud

<u>Services cloud : Ease of deployments</u>

Amazon, Google Cloud, Azure, Cloud Foundry, Alibaba

<u>Micro-services architecture</u>

Services de discovery, configuration server, load balancing, gateway, resilience and circuit breaker

Spring Cloud Contract : Design by contract

Observability

- Since version 3.x, SpringBoot relies heavily on ***Micrometer***.
  - Starters allow micrometer metrics to be published to visualization systems: ***DataDog, Dynatrace, Influx, Graphite, New Relic, Prometheus, Wavefront***
  - Other starters allow traceability of requests in micro-service architectures: ***Brave et Zipkin***

SpringBoot

Auto-configuration
Starters SpringBoot
Development SpringBoot
Configuration properties

Project structure

Recommendations :

- Locate the Main class in the root package
- Annotate it with :
  - Either the annotations
    - ***@EnableAutoConfiguration***
    - ***@ComponentScan***
    - ***@Configuration***
  - Or simply : ***@SpringBootApplication***

Typical structure

```
com
 +- example
     +- myproject
         +- Application.java
         |
         +- config
         |   +- SecurityConfig.java
         |   +- SwaggerConfig.java
         |
         +- domain
         |   +- Customer.java
         |   +- CustomerRepository.java
         |
         +- service
         |   +- CustomerService.java
         |
         +- rest
             +- CustomerRestController.java
```

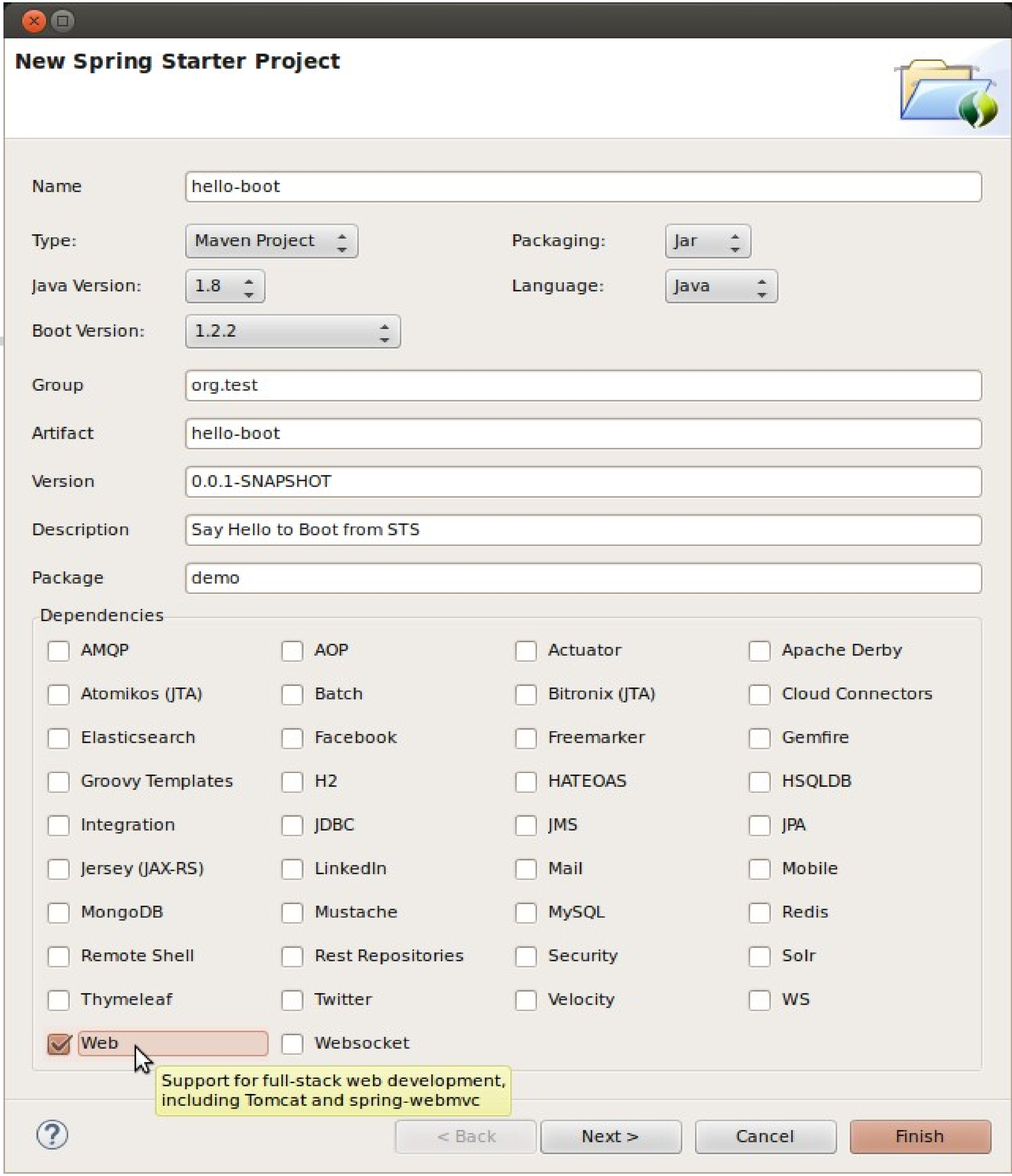
Spring Tool Suite

Distributed by Pivotal for VSCode, Atom or Eclipse

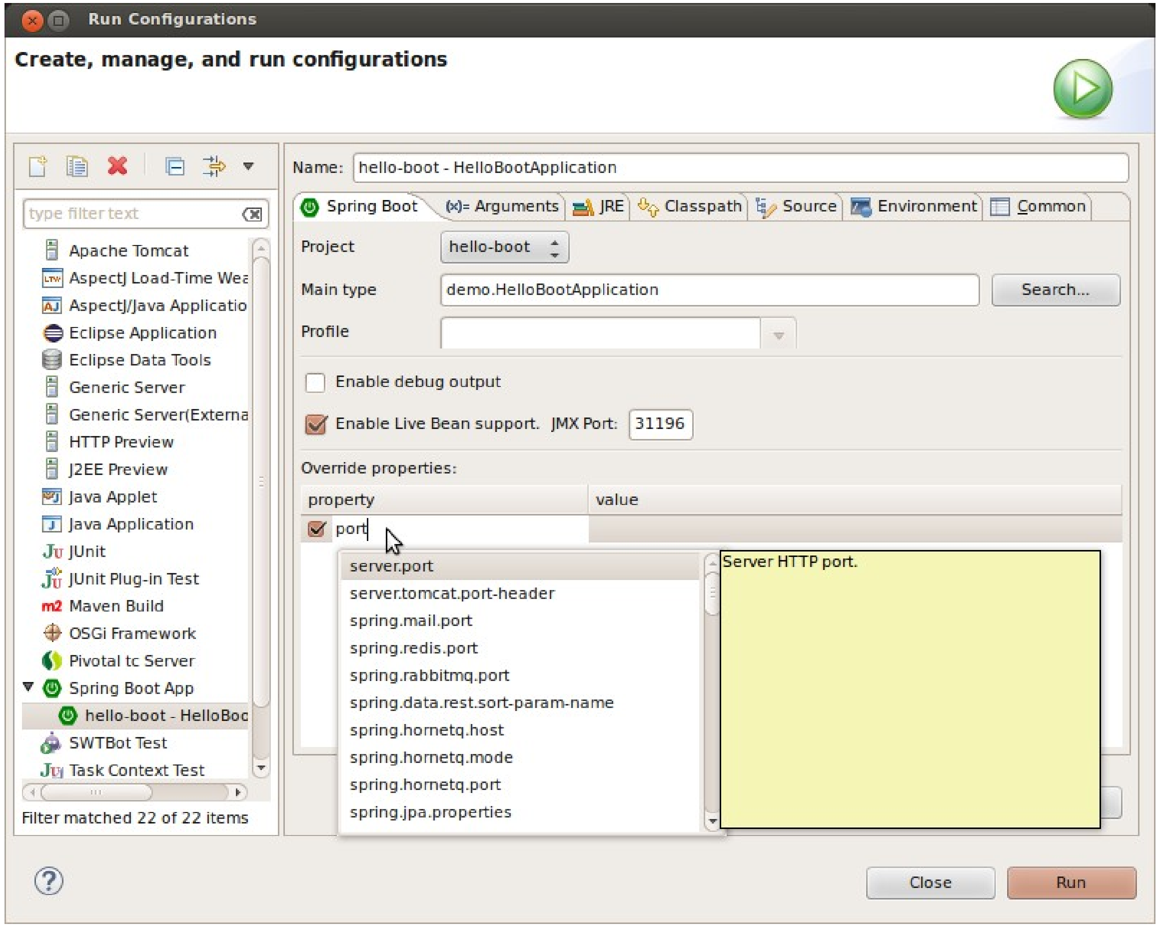
Integration exists with other IDEs

It offers mainly:

- **Creation project wizard** and starter edition connected directly to *Spring Initializr*
- **Auto-completion** on configuration properties
- **Boot Dashboard View** to start application
- **Run configurations screen** adapted to spring boot apps


New Spring Starter Project
Name
hello-boot
Type:
Maven Project
Packaging:
Jar
Java Version:
1.8
Language:
Java
Boot Version:
1.2.2
Group
org.test
Artifact
hello-boot
Version
0.0.1-SNAPSHOT
Description
Say Hello to Boot from STS
Package
demo
Dependencies
AMQP
AOP
Actuator
Apache Derby
Atomikos (JTA)
Batch
Bitronix (JTA)
Cloud Connectors
Elasticsearch
Facebook
Freemarker
Gemfire
Groovy Templates
H2
HATEOAS
HSQLDB
Integration
JDBC
JMS
JPA
Jersey (JAX-RS)
LinkedIn
Mail
Mobile
MongoDB
Mustache
MySQL
Redis
Remote Shell
Rest Repositories
Security
Solr
Thymeleaf
Twitter
Velocity
WS
Web
Websocket
Support for full-stack web development,
including Tomcat and spring-webmvc
< Back
Next >
Cancel
Finish

Run Configurations Spring Boot



Reloading code

- Because Spring Boot applications are a simple Java application, hot-code reloading must be supported.
  - => This eliminates the need to restart the application with each code change

Dev Tools

The ***spring-boot-devtools*** module can be added via a dependency

It provides :

- Adding configuration properties for development. Ex : *spring.thymeleaf.cache=false*
- Automatic restart when a class or the configuration change.
- *LiveReload Server* : Allow to reload the browser for web develoment

Logging

- Spring uses Common Logging internally but allows to choose its implementation

- Configurations are provided for:
  - Java Util Logging
  - Log4j2
  - Logback (default)

Format of traces

A line contains :

- Timestamp (ms)
- Severity level : ERROR, WARN, INFO, DEBUG or TRACE.
- Process ID
- A separator --- .
- Name of the thread enclosed with [].
- The name of the Logger <=> Name of the class.
- A message
- A note enclosed in []

Configure traces via Spring

By default, Spring displays ERROR, WARN, and INFO level messages on the console

- *java -jar myapp.jar* ***–debug*** : Activate DEBUG messages
- Properties ***logging.file*** and ***logging.path*** can specify a file to log
- Severity levels can be changed for each logger

```
logging.level.root=WARN
logging.level.org.springframework.web=DEBUG
logging.level.org.hibernate=ERROR
```

Lab : Set up IDE

SpringBoot

Auto-configuration
Starters SpringBoot
Development SpringBoot
Configuration properties

Configuration properties

- Spring Boot allows externalization of configuration properties:
  - You can use **properties** or **YAML** files, environment variables or command-line arguments.
- The property values are then injected into the beans:
  - Directly via ***@Value***
  - Or associate to a structured object via ***@ConfigurationProperties*** annotation

Order of precedence

The order of precedence for defining properties is as follows :

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 usually placed in the following locations :

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

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

N.B See annexe for YAML format

Filtered and random values

Spring 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}
```

Property injection : *@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(regexp = "\\d{3}-\\d{3}-\\d{4}")
  private String adminContactNumber;

  @Min(1)
  private int refreshRate;
    ..............
}
```

Profile-specific properties

Profile-specific properties (ex: integration, production) are specified differently depending on the format of the properties file.

- If using the .properties format, additional files can be provided: ***application-{profile}.properties***
- If you use the *.yml* format everything can be done in the same file

Example YAML

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

Activating Profiles

- Profiles are enabled by the property ***spring.profiles.active.***
- Typically via the command line :
  For example :
  `--spring.profiles.active=dev,hsqldb`

- Seceral profiles can be activated simultaneously

Persistence

Principles of SpringData

SpringData JPA

NoSQL

Introduction

- **Spring Data**'s mission is to provide a simple and consistent programming model for access to data regardless of the underlying technology (Relational, NoSQL, Cloud, Search Engine)
- Used with Spring Boot, a default configuration is available immediately and you can start coding without plumbing work

Advantages of SpringData

SpringData offers :

- An abstraction of the notion of **repository** and object mapping
- **Dynamic query generation** based on method naming rules
- Base implementation classes that can be used

Interfaces *Repository*

The central interface of Spring Data is Repository

(It's a marker class)

The interface takes arguments

- The **persistent class**
- Its **id**.

The sub-interface ***CrudRepository*** add all the CRUD methods

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

Interface *CrudRepository*

```
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.
}
```

Sample

```
public interface MemberRepository extends
  CrudRepository<Member, Long> {}
```

Just with this interface definition, Spring will create an implementation which already have all CRUD methods of CrudRepository

Query Methods

After extending the interface, it is possible to define methods for making queries

The requests to execute are deduced:

- From the ***name*** of the method
- From the ***@Query*** annotation

Using the name

When using the method name, these must be prefixed as follows:

- Query: *find*By**
- Count : *count*By**
- Deletion : *delete*By**
- Fetching *: get*By**

The first ******* can indicate a flag (as Distinct for example)

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

The rest is parsed and specifies the ***where*** clause and possibly ***orderBy***

Exemple

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

/**
 * Tous les membres 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 Optional<Member> fullLoad(Long id);
```

Where clause

- Expressions usually consist of properties of the entity combined with *AND* and *OR*

```
findByLastnameAndFirstname(…)
```

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

```
findByLastnameAndBirthDateGreaterThan(…)
```

- The *IgnoreCase* flag can be assigned individually to the 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

```
findByLastnameAndFirstnameOrderByLastNameDesc(…)
```

Parameters

In addition to the properties parameters, SpringBoot is able to recognize Pageable or Sort type parameters to dynamically apply pagination and sorting.

The return values can then be:

- *Optional<entity> , List<entity>*
- *Page* knows the total number of items by performing a count query,
- *Slice* only knows if there is a next page

```
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);
```

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 *@Query*

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

– Highest priority method

```
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

Principles of SpringData

SpringData JPA

NoSQL

Advantages of the starter

- spring-boot-starter-data-jpa provides the following dependencies:
  - Hibernate
  - Spring Data JPA .
  - Spring ORMs
- By default, all classes annotated with @Entity, are scanned and taken into account
- The starting location of the scan can be reduced 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(){}
```

Data source configuration / Reminders

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

A datasource encapsulates

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

Support for embedeed database

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

It is not necessary to provide a login 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 databases can also be auto-configured.

The properties required to configure are:

```
spring.datasource.url=jdbc:mysql://localhost/test
spring.datasource.username=dbuser
spring.datasource.password=dbpass
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

Configuration of the pool

Some properties configure the connections pool

For example :

```
spring.datasource.hikari.connection-timeout=10000
spring.datasource.hikari.maximum-pool-size=50
spring.datasource.hikari.minimum-idle= 10
```

Mode *create, create-drop*

Hibernate can be configured in *create*, *create-drop* mode

Which is very convenient for development and testing

The database is recreated at each startup and it is possible to execute a data initialization script (By default *import.sql* at the root of the classpath)

=> The database is in a known state at each startup

Properties

- Embedded JPA databases are created automatically.
- For others, it is necessary to specify the property ***spring.jpa.hibernate.ddl-auto***
  - 5 possible values: *none, validate, update, create, create-drop*
- Or the native Hibernate properties
  - They can be specifed with the prefix *spring.jpa.properties.**
    Ex :
    `spring.jpa.properties.hibernate.globally_quoted_identifiers=true`

Transactional behaviour

By default, CRUD methods are transactional.

For read operations, the transaction setup *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 then makes it possible to delimit a transaction for non-CRUD 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

Beans ***JdbcTemplate*** and ***NamedParameterJdbcTemplate*** are auto-configured and can be directly injected

Their behaviour can be customized with the properties : *spring.jdbc.template.**

Ex :

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

Exemple

```
@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 layers

You can also get the beans injected to allow you to code at a lower layer:

- At the JDBC level, by having the *DataSource* injected
- At the JPA level, by injecting the *entityManager* or the *entityManagerFactory*

Restful APIs

Spring MVC and RestFul APIs

RESTFul principles

Serialization with Jackson

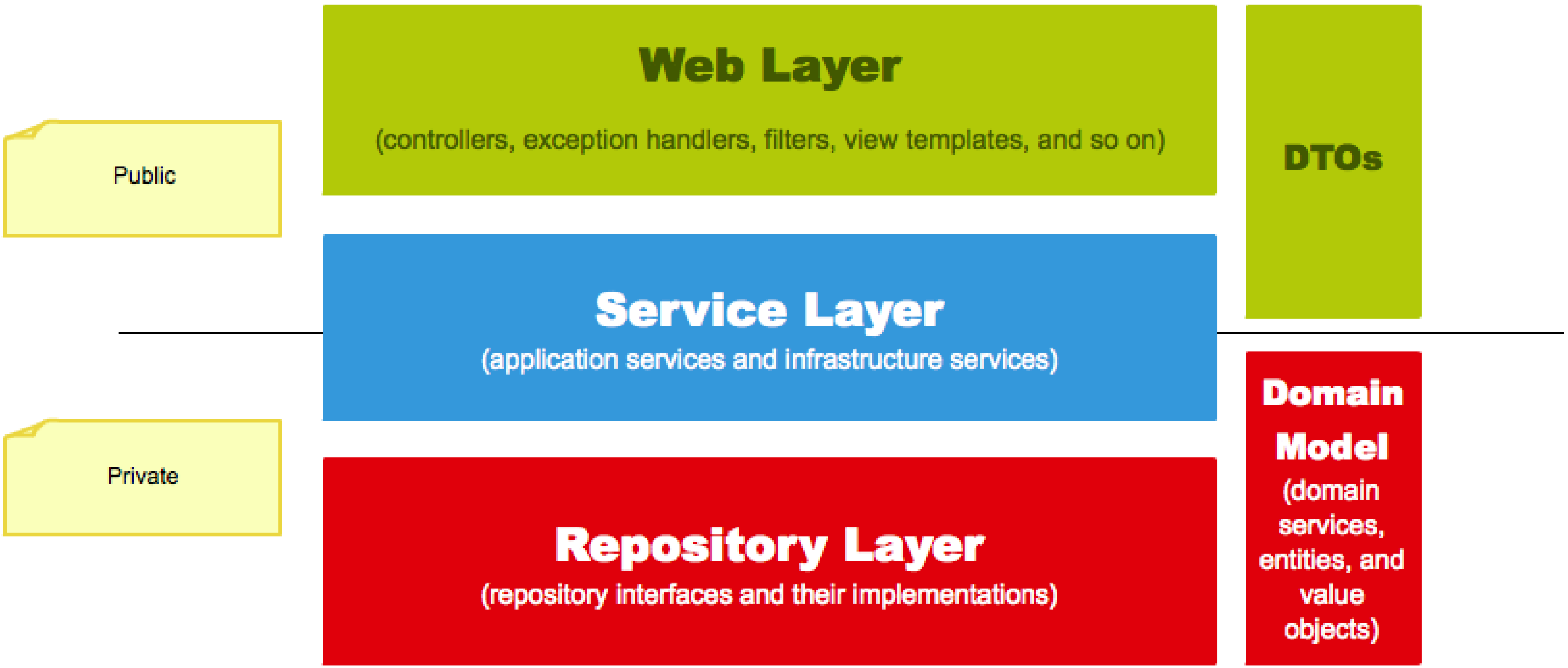
Exceptions, CORS and OpenAPI

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***

In the rest of the support, we focus on RestController

Typical layers of a RESTful sevice



@RestController

By default, controlers annotated with ***@RestController*** produced and consumed ***JSON***

- Return values of controller methods are serialized in JSON
- Request body are deserialized in Objects to fill the method arguments

@RequestMapping

@RequestMapping

- At the class level, indicates the basic path of all the resources of this controller
- At the method level,
  - ***path*** : Fixed value or URI template
  - ***method*** : To limit to a HTTP method
  - ***produce***/***consume*** : Precise the format of input/output

Declinaison of *@RequestMapping*

Spring provides annotations which limit to an http method :

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

It is also possible to limit to values of headers or parameters :

```
@GetMapping(path = "/pets", headers =
"myHeader=myValue", params =
"myParam=myValue")
```

Methods implementation

Annotations associate methods to URI and map arguments to element of the URI (path, parameter, ...)

The return of the method is either

- An **Data Object** (Entity or DTO), the object will be serialized by Jackson
- A ***ResponseEntity*** which encapsulates the HTTP response, then Response status, Headers, Cookies and Body can be finely precised

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

URI template

An URI template allow to define variables in the URI :

`http://www.example.com/users/{userId}`

A ***@PathVariable*** annotation can then be used to associate this variable to an argument of a method

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

@RequestParam

@RequestParam allow to associate an method argument to a request parameter. Conversion is automatically performed by Spring

```
@Controller
@RequestMapping("/pets")
@SessionAttributes("pet")
public class EditPetForm {

    // ...

    @GetMapping
    public String setupForm(@RequestParam("petId") int petId, ModelMap model) {
        Pet pet = this.clinic.loadPet(petId);
        model.addAttribute("pet", pet);
        return "petForm";
    }

    // ...

}
```

Arguments Validation

- Arguments of method must be validated
- Typically, if they are not valid a proper error response should be return :
  - A clear message indicating what went wrong? Which field has an error and what are the accepted values? What the consumer can do to fix the error?
  - Proper Response Status : 400 Bad Request.
- With Spring Boot, it is very straightforward to have this behaviour :
  - Annotate with ***javax.bean.validation*** annotations the model
  - Precise ***@Valid*** on the argument of the method which must be validated

Example *@RequestBody*

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

    // ...

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

        return pet;
    }

    // ...

}
```

Types des valeurs de retours des méthodes

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

- A **Model or DTO** class which will be converted to JSON via the Jackson library.
  The default return code is then 200
- A **ResponseEntity<T>** object allowing you to position the desired return codes and HTTP headers
- It is also possible to annotate the method with ***@ResponseStatus*** to indicate the HTTP return code

Exemples

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

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

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

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

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public User register(@Valid @RequestBody User user) {

      return userRepository.save(user);

    }

}
```

Restful APIs

Spring MVC and RestFul APIs

RESTFul principles

Serialization with Jackson

Exceptions, CORS and OpenAPI

Auto-descrption

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 UBER company API resource:

GET https://api.uber.com/<version>/partners/101/payments

=> A developer intuitively knows that he will find in this resource the information regarding the payments received by the partner UBER (independent driver).

Core principles

A ressource <=> An entity

URI	GET	PUT	POST	DELETE
Collections : *http://api.example.com/produits*	List all products	Replaces the product list with another list	Creates a new entry in the collection.	Deletes the collection.
Element, *http://api.example.com/produits/5*	Retrieves product ID 5	Replaces or creates the element	Treat the element as a collection and add an entry to it	Remove item

Naming rules

For objects linked 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

Method arguments

Path parameter : Used for required arguments (id for example)

Query parameter : Used for optional arguments (q, page, size, sort, ...)

Header : Mainly security

Body : Structured Data (Entity or DTO)

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*** : Client error
- ***5xx*** : Serveur error

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 not the HTTP method
- ***406 – Not Acceptable*** : The 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 exceeded

Restful APIs

Spring MVC and RestFul APIs
RESTFul principles
Serialization with Jackson
Exceptions, CORS and OpenAPI

JSON serialization

- One of the main issues with Spring back-ends 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 behaviour

```java
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 usually done by ***ObjectMappers***

```
// Sérialisation
Member m = memberRepository.findById(4l) ;
ObjectMapper objectMapper = new ObjectMapper() ;
String jsonString = ObjectMapper.writeValueAsString(m) ;
…
// Désérialisation
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 his needs, 3 alternatives:

- Create specific DTO classes.
  The Service layer transforms the Entity classes coming from the Repository layer into Data Transfer Object classes encapsulating the data that is serialized by Jackson
- Use the annotations provided by Jackson
  On DTO classes or Entity classes, use Jackson annotations to adapt to the need for serialization
- Use *@JsonView* annotation
  The same Entity or Dto object can then be serialized differently depending on the use case
- Implement its own Serializer/Deserializer.
  Spring provides *@JsonComponent*

Example DTO

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

    UserDto retreiveUser(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;
    }
}
```

Date format

To have a String representation of the dates according to the requirements of the front-ent, one solution 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;
}
```

Bi-directional Relations

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

Bi-directional Relations
One solution

By 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*

Bi-directional Relations
Another solution

By 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 { ... }
```

Sérialisation d’un Item :

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

Bi-directional Relations
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

Inheritance relationships can be defined in empty static classes

```
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 the 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?

https://mkyong.com/java/jackson-jsonview-examples/

@JsonView

Model Class Annotations

```
..
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;
```

https://mkyong.com/java/jackson-jsonview-examples/

Activating a View

```
@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’s annotations

- ***@JsonProperty, @JsonGetter, @JsonSetter, @JsonAnyGetter, @JsonAnySetter, @JsonIgnore, @JsonIgnorePoperty, @JsonIgnoreType*** : Allow to set JSON properties
- ***@JsonRootName*** : Tree JSON
- ***@JsonSerialize, @JsonDeserialize*** : Indicates specialized de/serializers
- ….

Specific serializer

- The Spring ***@JsonComponent*** annotation allows to register Jackson serializers/deserializers
- It must be placed on implementations of JsonSerializer and JsonDeserializer or on classes containing inner-class of this type

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

OpenInView

- Lors d'une application Web, Spring Boot enregistre par défaut l'intercepteur *OpenEntityManagerInViewInterceptor* afin d'appliquer le pattern "***Open EntityManager in View***" permettant d'éviter les *LazyException* dans les vues
- Si ce n'est pas le comportement voulu :
  `spring.jpa.open-in-view = false`

Restful APIs

Spring MVC and RestFul APIs
RESTFul principles
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.
- With a Rest API, a method allows you to configure the CORS[1]

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

Voir : https://spring.io/guides/gs/rest-service-cors/

Errors handling

- 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:
  - REST
    - 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

```java
@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, DocumentNotFoundException.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 UI : 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`

Services Interactions

RestClient

Messaging

Restful call

- Spring4 provides the ***RestTemplate*** class to facilitate calls to REST services.
- Spring Boot does not provide a self-configured bean of type RestTemplate but it self-configures a ***RestTemplateBuilder*** to create them
- *RestTemplate* is going to ge deprecated

Example

```
@Service
public class MyBean {
private final RestTemplate restTemplate;

public MyBean(RestTemplateBuilder restTemplateBuilder) {
 this.restTemplate =
   restTemplateBuilder.basicAuthentication("user","password")
                     .build();
}
public Details someRestCall(String name) {
  return this.restTemplate.getForObject("/{name}/details",
                                        Details.class,
                                        name);
 }
}
```

WebClient

- ***WebClient*** is the new interface provided by Spring Webflux to perform web requests.
- The solution offers support for synchronous and asynchronous interactions, so it can be used on both web stacks (servlet and reactive)
- The interface has a single implementation: *DefaultWebClient*

Creating a WebClient

3 alternatives to create a WebClient

```
// Config par défaut
WebClient client = WebClient.create();

// Base Uri
WebClient client = WebClient.create("http://localhost:8080");

// Builder
WebClient client = WebClient.builder()
  .baseUrl("http://localhost:8080")
  .defaultCookie("cookieKey", "cookieValue")
  .defaultHeader(HttpHeaders.CONTENT_TYPE, MediaType.APPLICATION_JSON_VALUE)
  .defaultUriVariables(Collections.singletonMap("url",
   "http://localhost:8080"))
  .build();
```

Preparing the request

The preparation of the request consists of specifying the HTTP method, the URL, the body and the headers.

```
client.post()
        .uri("/resource")
        .bodyValue("data")
        .header(HttpHeaders.CONTENT_TYPE, MediaType.APPLICATION_JSON_VALUE)
        .accept(MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML)
```

Retreiving the response

To retrieve the response, you can use ***exchangeToMono*** and ***exchangeToFlux*** which allow you to inspect the response (header, status code,)

Or simply ***retrieve*** which allows to retrieve the body of the response

Examples

```
// exchangeToMono
Mono<String> response = headersSpec.exchangeToMono(response -> {
  if (response.statusCode().equals(HttpStatus.OK)) {
      return response.bodyToMono(String.class);
  } else if (response.statusCode().is4xxClientError()) {
      return Mono.just("Error response");
  } else {
      return response.createException()
        .flatMap(Mono::error);
  }
});


// Retreive simple
Mono<String> response = headersSpec.retrieve()
  .bodyToMono(String.class);
```

Services Interactions

RestClient

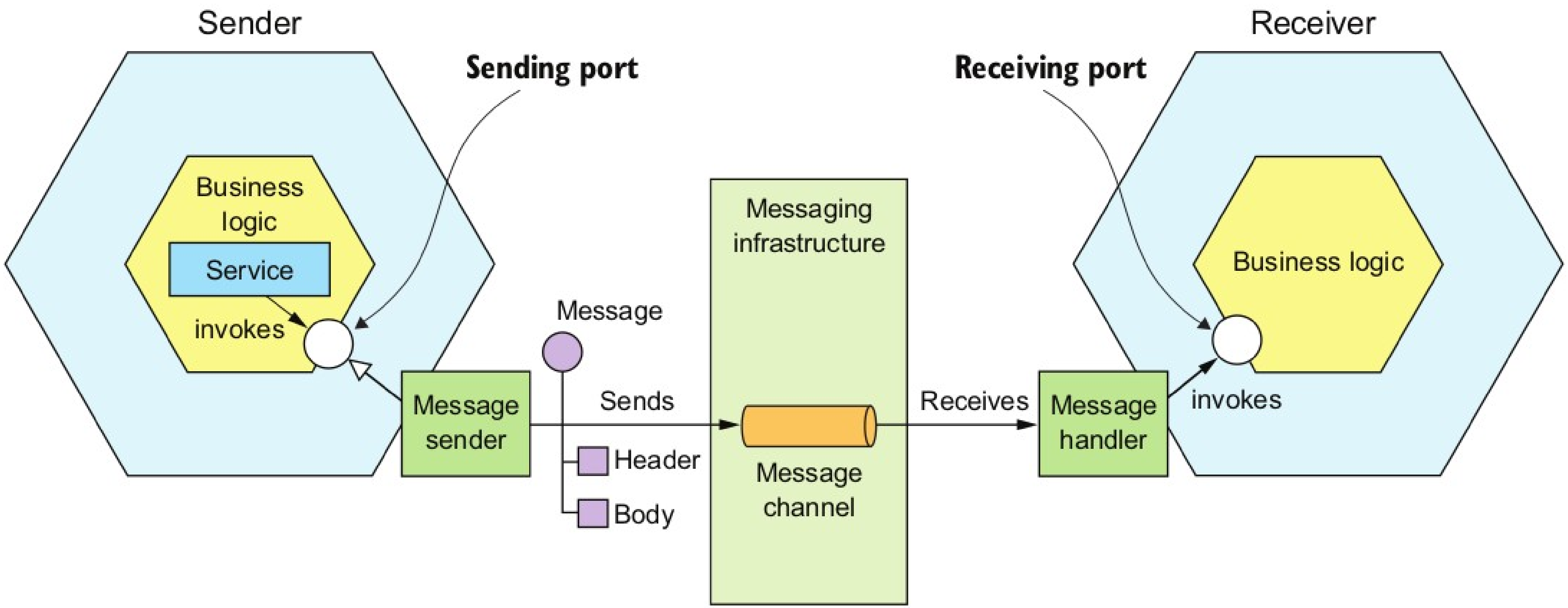
Messaging

Introduction

- Asynchronous communications between processes provide several advantages:
  - Decoupling of message producer and consumer
  - Scaling
  - Implementation of Saga micro-services patterns[1], Event-sourcing Pattern[2]
- Some challenges
  - 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*

Architecture



Message semantic

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

There are 3 types 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 indicating that something has just happened. Often a business event

Channels

2 types 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

All 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 expect a response
- Publish and Subscribe :
  The producer does not expect an answer
- Publish and asynchronous responses
  Producer is notified when responses arrive

API Specification

- The specification consists of defining
  - Channel names
  - Types of messages and their format. (Typically JSON)
- Unlike REST and OpenAPI, there is no standard

Message Broker

- A message broker is an intermediary through which all messages pass
  - 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

Offre Spring

Messaging starters for:

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

Event processing pipeline:

- Kafka Stream

Event-driven microservices architecture

- Spring Cloud Stream
- Spring Data Flow

Example *spring-kafka*

Sending a 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 a message :

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

Spring Security

Principles

Stateful/stateless models

Auto-configuration Spring Boot

OpenIdConnect and oAuth2

Spring Security

Spring Security mainly handles 2 aspects of security:

- **Authentication**: Ensuring User or System Identity
- **Authorization**: Verify that the user or system has access to a resource.

Spring Security makes it easy to implement security on Java applications by

- integrating authentication providers:
  - Custom
  - Or integrating with standards (LDAP, OpenID, Kerberos, PAM, CAS, oAuth2)
- allowing the configuration of access constraints to URLs and methods of business services

Principles and mechanism

Default web security configuration can be caused by *@EnableWebSecurity* annotation or by SpringBoot

- The ***springSecurityFilterChain*** bean encapsulates a chain of filters intercepting all HTTP requests. Each filter is responsible for one aspect of security. The filter chain is highly configurable and adapts to all approaches

If you also want to add security at the method level, you must explicitly activate it (even in a SpringBoot context) with ***@EnableGlobalMethodSecurity***

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 (e.g. in http session)*

Security Customization

Configuration customization consists of:

– To customize the springSecurityFilterChain filter by creating a bean of type ***SecurityFilterChain***
the HttpSecurity class is a builder facilitating its creation

– To customize authentication:

- by creating an ***AuthenticationManager*** type bean
The AuthenticationManagerBuilder class facilitates the creation of Realm (inMemory, jdbc, ldap, ...)
- or completely customized by implementing a bean ***UserDetailsService***

– Ignoring security for some resources by defining a Bean of type ***WebSecurityCustomizer***: lambda taking a *WebSecurity* object as an argument

https://spring.io/blog/2022/02/21/spring-security-without-the-websecurityconfigureradapter

Example SecurityFilterChain

```
// Ex : Spring MVC
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws
   Exception {
  http
    .authorizeRequests(authorize ->
       authorize.anyRequest().authenticated())
    .formLogin(withDefaults())
    .httpBasic(withDefaults());
     return http.build();
}
```

Example : SecurityWebFilterChain

```
// Ex : Spring WebFlux
@Bean
public SecurityWebFilterChain securityWebFilterChain(
ServerHttpSecurity http) {
  return http.authorizeExchange()
         .pathMatchers("/actuator/**").permitAll()
         .pathMatchers("/auth/**").permitAll()}
         .anyExchange().authenticated()
         .and()
         .oauth2Login().csrf().disable().build();
}
```

Debugging security

To debug the configuration:

– Check the log which displays all the filters of *springSecurityFilterChain*

To debug exacution :

– Activate DEBUG traces:

```
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);
}
```

Spring Security

Principles

Stateful/stateless models

Auto-configuration Spring Boot

OpenIdConnect and oAuth2

Web App and Rest API

- Web applications (stateful) and REST APIs (stateless) do not 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

Stateful authentication

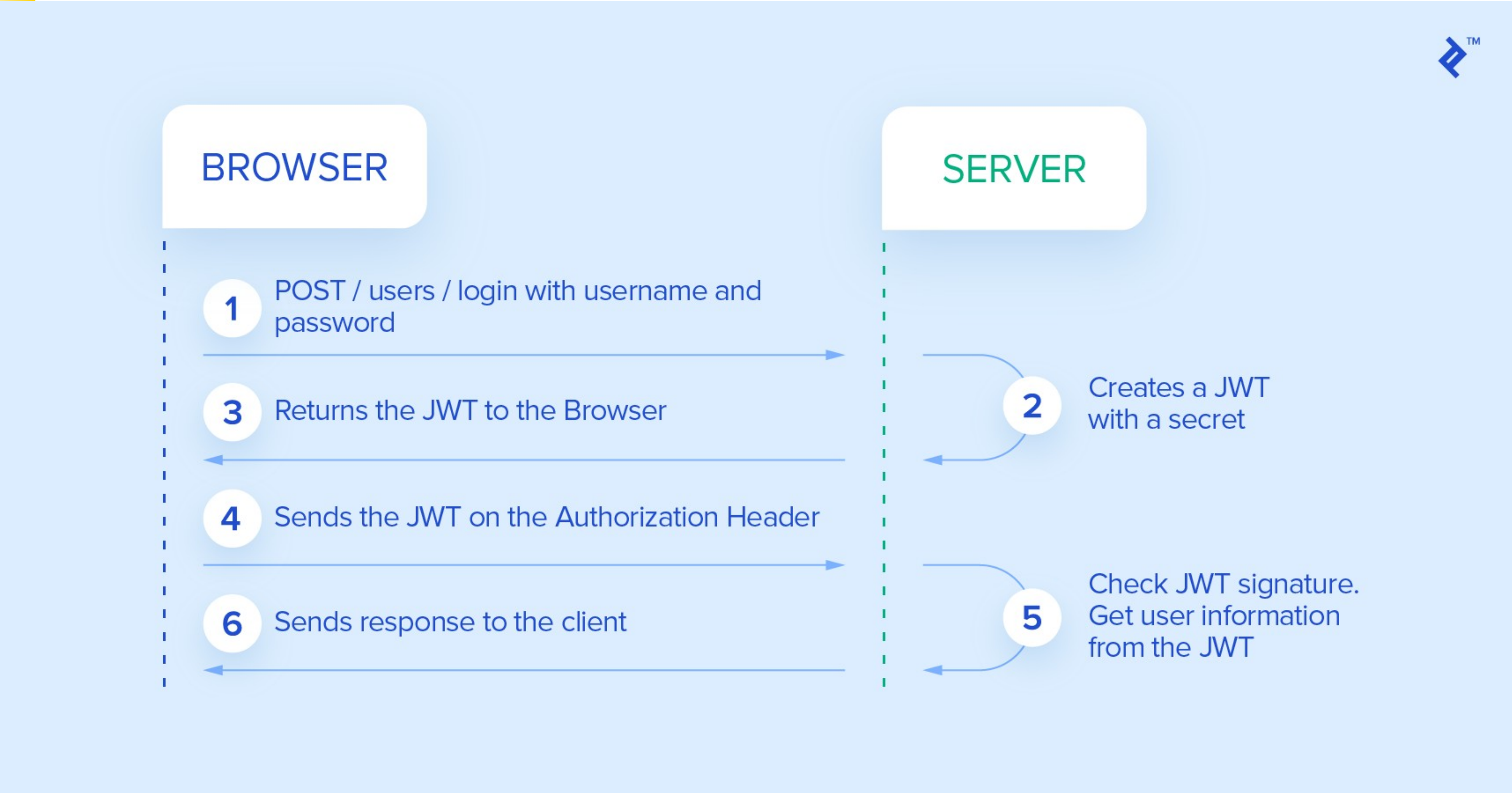
Rest standard or monolithic web app

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. It is recoverable at any time by ***SecurityContextHolder.getContext().getAuthentication()***

Stateless authentication
API REST - microservice

1. The client requests a protected resource.
2. The server returns a response indicating that one must authenticate 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 decrypts the token and derives the user's rights. It authorizes or prohibits access to the resource

Sample stateless security



Spring Security

Principles

Stateful/stateless models

Auto-configuration Spring Boot

OpenIdConnect and *oAuth2*

SpringBoot support

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

Default Authentication Manager

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

Properties can be changed via *application.properties* and the *security* prefix.

```
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)

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

Spring Security

Principles
Stateful/stateless models
Auto-configuration Spring Boot
OpenIdConnect* and *oAuth2

Protocol Roles

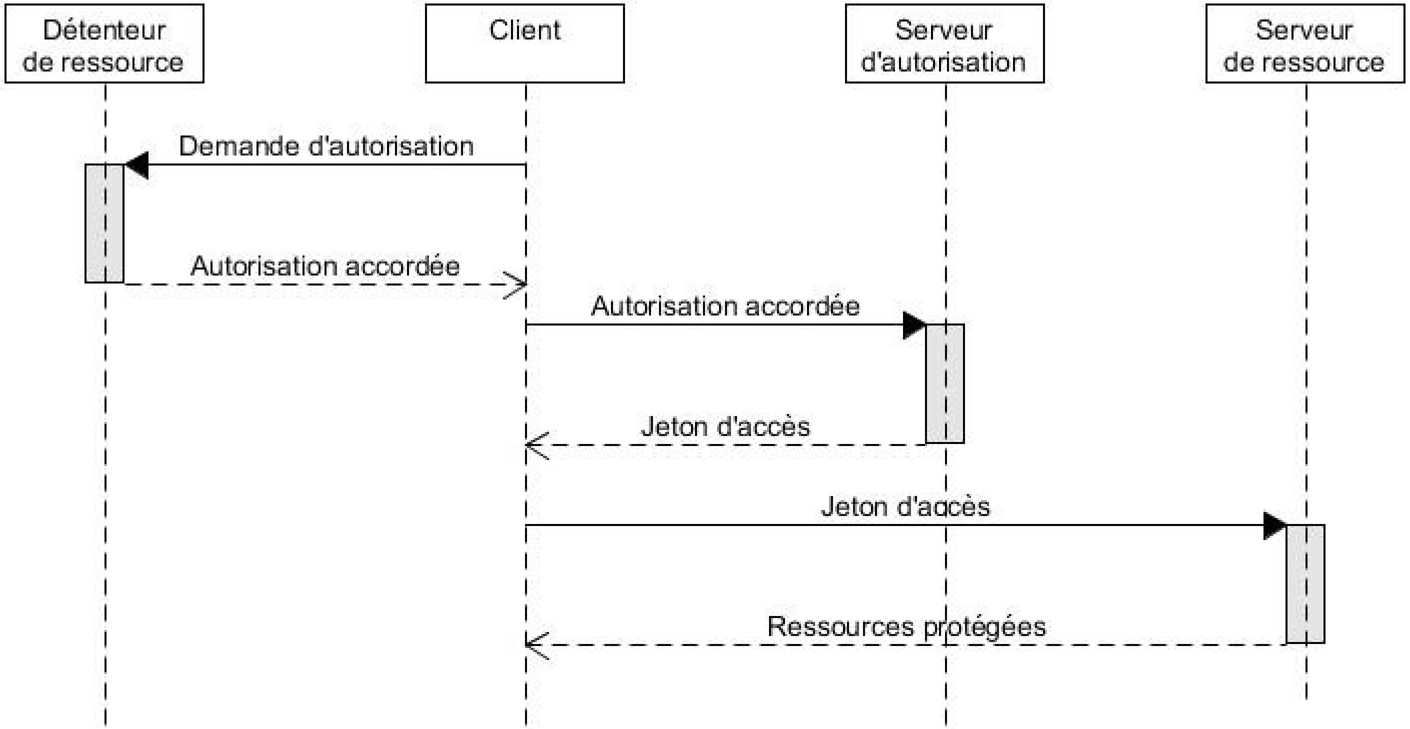
The ***Client*** is the application trying to access the user account. It needs to get permission from the user to do so.

The ***resource server*** is the API used to access protected resources

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

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

NB: A protocol participant can play several roles


Détenteur de ressource
Client
Serveur d'autorisation
Serveur de ressource
Demande d'autorisation
Autorisation accordée
Autorisation accordée
Jeton d'accès
Jeton d'accès
Ressources protégées

Scénario

1. Pre-register the client with the authorization service (=> client ID and a secret)
2. Obtain user permission. (4 types of grants)
3. Obtaining the token (expiration date)
4. Call of the API to obtain the desired information 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 present 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

Client registration

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

Customer must provide:

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

The server responds with:

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

Scope

- 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

OAuth2 Grant Type

Different means for the user to give his consent : **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 provided directly. Some servers forbid this mode
- ***password :*** The client provides the user's credentials
- ***client credentials*** : The client is the user
- ***device code*** :

Token usage

- The token is typically transmitter with the Authorization header

```
GET /profile HTTP/1.1

Host: api.example.com
Authorization: Bearer MzJmNDc3M2VjMmQzN
```

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

Token Validation

When receiving the token, the resource server must validate the authenticity of the token and extract its information different techniques are possible

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

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, ...) or the starter ***spring-security-oauth2-jose***

SpringBoot support

Support for oAuth through Spring 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
- There is no longer support for an authorization server

3 starters are now provided:

- ***OAuth2 Client*** : Iintegration to use an oAuth2 login provided by Google, Github, Facebook, …
- ***OAuth2 Resource server*** :Application allowing to define ACLs in relation to client scopes and roles contained in oAuth tokens
- ***Okta*** : To work with oAuth provider Okta

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*

```yaml
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 Setup 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

SpringBoot and tests

Spring Test

Spring Boot support

Auto-configured tests

Versions Spring/SpringBoot/JUnit

SpringBoot 1, Spring 4, JUnit4

Latest release September 2018

SpringBoot 2, Spring 5, JUnit5

First version ~2018

SpringBoot 3, Spring 6, JUnit5

2023

spring-test

Spring Test brings little for unit testing

- ***Mocking*** the environment especially the servlet or Reactive API
- Utility packages : *org.springframework.test.util*

And plenty for integration testing (involving a Spring ApplicationContext):

- Caching the Spring context to speed up testing
- Injection of test data
- Transaction management (roll-back)
- utility classes
- JUnit4 and JUnit5 integration

Integration JUnit

- JUnit4 :

`@RunWith(SpringJUnit4ClassRunner.class)`

or `@RunWith(SpringRunner.class)`

Allows you to load a Spring context, perform dependency injection, etc.

- JUnit5 :

`@ExtendWith(SpringExtension.class)`

Also allows you to load a Spring context, perform dependency injection, etc.

And in addition to dependency injection for test methods, runtime conditions based on Spring configuration, additional annotations to manage transactions

Example JUnit5

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


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

SpringBoot and tests

Spring Test
Spring Boot support
Auto-configured 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 which links Spring Test to Spring Boot*
- ***Spring Boot Test Autoconfigure*** *: Auto-configured tests*
- *JUnit4, AssertJ, Hamcrest* (SB 1.x)or JUnit5 (SB 2.X):
- *Mockito* : A framework for generating Mock classes
- *JSONassert, JsonPath* : A library for JSON assertions

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

- It is possible to use the ***@SpringBootTest*** annotation replacing the standard spring-test configuration (*@ContextConfiguration*)
- The annotation creates the application context (*ApplicationContext*) used when testing using SpringApplication (main class)

Equivalence

```
// Annotations SpringBootTest
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment=WebEnvironment.RANDOM_PORT)
public class SpringBootWebApplicationTests {


// Standard Annotations
@RunWith(SpringRunner.class)
@SpringApplicationConfiguration(classes =
  SprintBootWebApplication.class)
@WebAppConfiguration
public class SpringBootWebApplicationTests
```

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 (servlet container is not started): *WebApplicationContext*
  - ***RANDOM_PORT*** : Loads a *ServletWebServerApplicationContext*. The container is started on a random port
  - ***DEFINED_PORT*** : Loads a ServletWebServerApplicationContext. The container is started on a specified port
  - ***NONE*** : No servlet environment.

Detection of the configuration

The ***@*Test*** annotations serve as a starting point for configuration research.

In the case of *SpringBootTest*, if the class attribute is not filled in, the algorithm searches for the first annotated class *@SpringBootApplication* ou *@SpringBootConfiguration* going up from packages

=> It is therefore recommended to use the same package hierarchy as the main code

Mocking

- The ***@MockBean*** annotation defines a Mockito bean
- This allows replacing or creating new beans
- Annotation can be used:
  - On test classes
  - 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");
    }

}
```

SpringBoot and tests

Spring Test
Spring Boot support
Auto-configured tests

Auto-configured tests

- 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");
    }

}
```

Spring MVC 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 server
- For *Selenium* or *HtmlUnit* testing, a web client is also provided

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
  - Possibility to change this behavior via *@Transactional*
- A *TestEntityManager* can be injected as well as a *JdbcTemplate*

Example

```java
@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 auto-configured tests

@WebFluxTest : Testing Spring Webflux Controllers

@JdbcTest : Only the 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 : Annotate a method

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

@WithSecurityContext : Which allows you to create the SecurityContext you want

Toward production

Monitoring with actuator

Deployment

Actuator

- Spring Boot Actuator provides support for monitoring and managing SpringBoot applications
- It can rely
  - On HTTP endpoints (If we used Spring MVC)
  - On JMX
- Activation of Actuator requires ***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 endpoints:

- ***beans*** : A list of Spring beans
- ***env*** / ***configprops*** : Liste of configuration properties
- ***health*** : Health of the app
- ***info*** : Arbitrary Information. In general, Commit, version
- ***metrics*** : Performance metrics
- ***mappings*** : Liste of configured endpoints
- ***trace*** : Trace of latest requests
- ***docs*** : Documentation
- ***logfile*** : Content of the log

If you develop an Endpoint-type Bean, it is automatically exposed via JMX or HTTP

Configuration

- Endpoints can be configured by properties.
- Each endpoint can be
  - Enabled/disabled
  - Secured by Spring Security
  - Mapped to custom URL
- In SB 2.x, only /health and /info endpoints are enabled by default
- To enable others endpoints:
  - *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.
  - It 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
```

Provided indicators

The SpringBoot starters provides their own health indicators:

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

Info endpoint

The default /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

Le endpoint ***metrics*** provides different metrics :

- System : Memory, Heap, Threads, GC
- Data sources: Active connections, pool status
- Cache : Size, Hit and Miss Ratios
- Tomcat Sessions
- ...

Endpoints of SpringBoot 2

/auditevents : List security events (login/logout)

/conditions : Autoconfiguration report

/configprops – Beans annotated by *@ConfigurationProperties*

/flyway ; DB Flyway Migration Information

/liquibase : DB Liquibase migrations

/logfile : logs

/loggers : Display and update log configuration

/scheduledtasks : Scheduled tasks

/sessions : HTTP sessions

/threaddump : Thread dumps

Toward 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 execute

```
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
```

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 creation

```
<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

Updating a Procfile :

```
web: java -Dserver.port=$PORT -jar target/demo-0.0.1-SNAPSHOT.jar
```

```
git push heroku master
```

Annex

Auto-configuration mechanism

NoSQL

Spring MVC

Spring Reactive

Auto-configuration

- Spring Boot's auto-configuration mechanism is based on:
  - Classical ***@Configuration*** classes which definebeans
  - ***@Conditional*** annotation which specify the conditions required to activate the configuration
- When Spring Boot starts, 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, @ConditionalOnMissingClass***
- The presence or absence of a bean : ***@ConditionalOnBean, @ConditionalOnMissingBean***
- On the value of a property : ***@ConditionalOnProperty***
- The presence of a ressource : ***@ConditionalOnResource***
- Whether the app is a web app or not: ***@ConditionalOnWebApplication*** ou ***@ConditionalOnNotWebApplication***
- An SpEL expression

Apache SolR example

```
@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 SolR starter pulls

- Conditional Configuration Classes
- SolR 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;
}
```

Annex

Auto-configuration

NoSQL

Spring MVC

Spring Reactive

Introduction

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

Ex : MongoDB

`spring-boot-starter-data-mongodb`

Connection to MongoDB

- SpringBoot automatically creates a bean ***MongoDbFactory*** which connects to *mongodb://localhost/test*
- The property ***spring.data.mongodb.uri*** can be used to specify another URL

Entity

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

A class of domain can be:

```
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

- SpringData also offers repository implementations
  - Just have the right dependencies in the class-path : *spring-boot-starter-data-mongodb*
- The example for JPA is then also valid in this environment

Usage

```
@Controller
public class MyControler {

@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 bean *MongoTemplate* is also auto-configured.
  - It can be injected .
- It is this class that implements the methods of the Repository interface behind the scenes but it can also be used directly.

Embededd Mongo

- It is possible to use an embedded Mongo
- Just have dependencies to :
  `de.flapdoodle.embed:de.flapdoodle.embed.mongo`
- The port used is either randomly determined or fixed by the property: `spring.data.mongodb.port`
- The traces of MongoDB are visible if *slf4f* is in the classpath

Annex

Auto-configuration
NoSQL
Spring MVC
Spring Reactive

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***

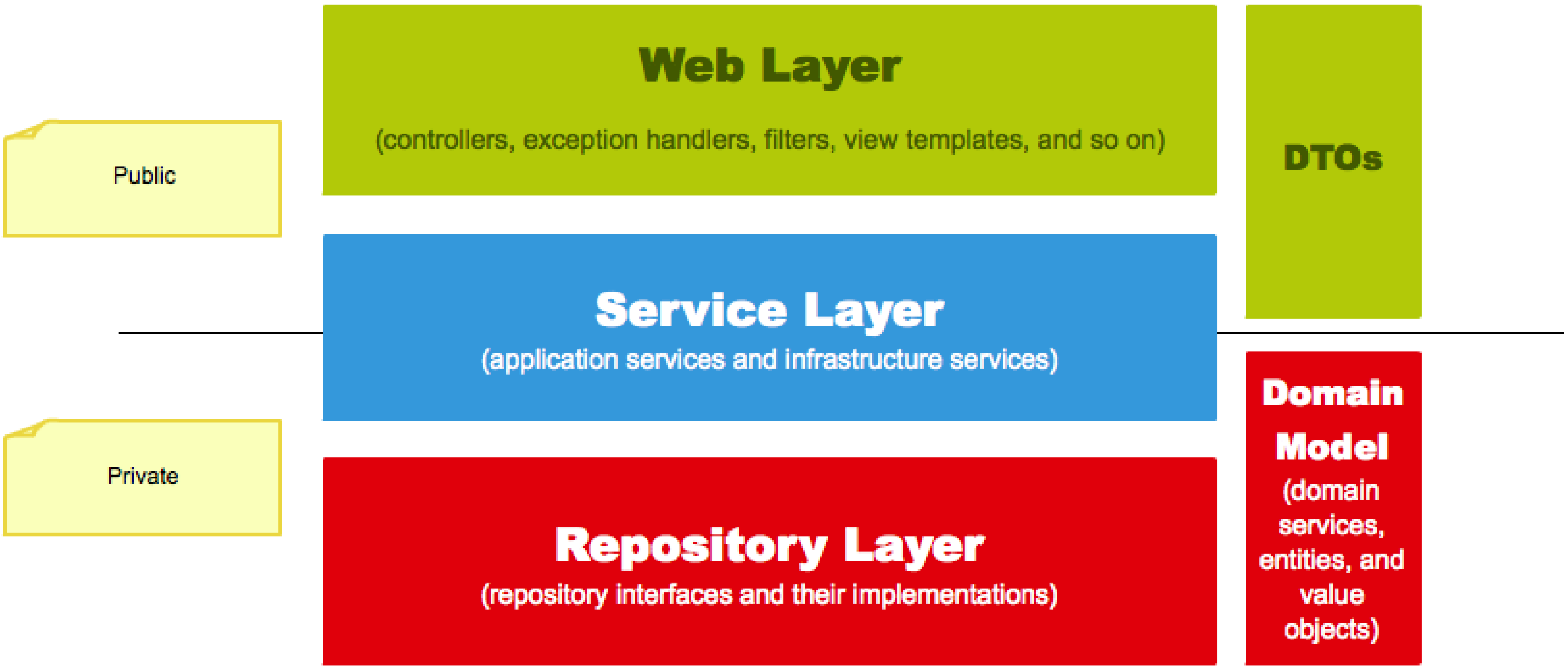
Model View Controller

The Model-View-Controller (MVC) framework is designed around the ***DispatcherServlet*** servlet which dispatches requests to controllers

– The controller/request mapping is done through the ***@RequestMapping*** annotation

  - Legacy controllers are responsible for preparing model data through ***Map-like*** interfaces.
    The request is then forwarded 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

Classic layered architecture



@Controller, @RestController

The ***@Controller, @RestController*** annotations annotate simple POJOs whose public methods are generally accessible via HTTP

```
@Controller
public class HelloWorldController {

    @RequestMapping("/helloWorld")
    public String helloWorld(Model model) {
        model.addAttribute("message", "Hello World!");
        return "helloWorld";
    }
}
```

@RequestMapping

@RequestMapping

- At class level indicates that all handler methods will be relative to this path
- At method level, the annotation specifies:
  - ***path*** : Fix value or URI template
  - ***method*** : To narrow to a HTTP method
  - ***produce***/***consume*** : Precise formats (mime-type) of input and output data

Complements *@RequestMapping*

Variants to limit to one method:

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

Limit to the value of a parameter or a header:

```
@GetMapping(path = "/pets", headers = "myHeader=myValue",
params = "myParam=myValue")
```

Use regular expression

```
@GetMapping(value = "/ex/bars/{numericId:[\\d]+}")
```

Use configuration properties

```
@RequestMapping("${context}")
```

Method Argument Types

A method of a controller can take arguments of type:

- The HTTP request or response (ServletRequest, HttpServletRequest, spring.WebRequest, ...)
- The HTTP session (HttpSession)
- The locale, the time zone
- Input/output streams
- The HTTP method
- The user authenticated by HTTP (Primary)
- A Map, org.springframework.ui.Model or org.springframework.ui.ModelMap representing the model exposed to the view
- Errors or validation.BindingResult: Errors from a previous form submission

If the argument is of another type, it requires **annotations** so that Spring can perform the necessary conversions from the HTTP request

Argument Annotations

Argument annotations allow you to associate an argument with a value in the HTTP request:

- ***@PathVariable***: Part of the URI
- ***@RequestParam***: An HTTP parameter
- ***@RequestHeader***: A header
- ***@RequestBody***: Request content using an HttpMessageConverter
- ***@RequestPart***: Part of a multi-part request
- ***@SessionAttribute***: A session attribute
- ***@RequestAttribute***: A request attribute
- ***@ModelAttribute***: A model attribute (request, session, etc.)
- ***@Valid***: Ensures constraints on the argument are valid

URI template

A URI template is used to define variable names:
`http://www.example.com/users/{userId}`

The ***@PathVariable*** annotation associates the variable with a method argument

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

Complements

- A @PathVariable argument can be of **simple type**, Spring does the conversion automatically
- If *@PathVariable* is used on a **Map<String, String>** argument, the argument is populated with all the variables in the template
- A template can be constructed from the combination of class and method annotations

@RequestParam

```java
@Controller
@RequestMapping("/pets")
@SessionAttributes("pet")
public class EditPetForm {

    // ...

    @GetMapping
    public String setupForm(@RequestParam("petId") int petId, ModelMap model) {
        Pet pet = this.clinic.loadPet(petId);
        model.addAttribute("pet", pet);
        return "petForm";
    }

    // ...

}
```

Types of method return values

For the MVC pattern:

- *ModelAndView*, *Model*, *Map*
- *Views determined by* : *View*, *String*

void : If the controller generated the response itself

For the REST Model:

- A *Model* or *DTO* class converted via an HttpConverter (REST JSON) that provides the HTTP response body
- A *ResponseEntity*<> allowing to position return codes and HTTP headers

Input/output formats

It is also to specify a list of media type allowing to filter on the Content-type header of the HTTP request

In input, specifies the expected format

```
@PostMapping(path = "/pets", consumes = "application/json")
public void addPet(@RequestBody Pet pet, Model model) {
```

Or output, specify the generated format:

```
@GetMapping(path = "/pets/{petId}",
  produces = MediaType.APPLICATION_JSON_UTF8_VALUE)
@ResponseBody
public Pet getPet(@PathVariable String petId, Model model)
  {
```

@RequestBody and converter

The @RequestBody annotation uses HTTPMessageConverters that rely on the content-type header of the request

- *StringHttpMessageConverter*
- *FormHttpMessageConverter (MultiValueMap<String, String>)*
- *ByteArrayHttpMessageConverter*
- *MappingJackson2HttpMessageConverter* : JSON
- *MappingJackson2XmlHttpMessageConverter* : XML
- ...

Spring MVC

Spring MVC basics

Spring Boot and Spring MVC

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

Bean’s Customization

- Add a ***WebMvcConfigurerAdapter*** type bean (or extend ***WebMvcConfigurer***) and implement the desired methods:
  - MVC configuration (ViewResolver, ViewControllers), exception handler or interceptors, Cors, ..
- If you want to customize SpringMVC core beans, declare a bean of type ***WebMvcRegistrations*** and register your own mapping handlers or exception resolvers

Exemple : Définition de *ViewController*

```
@Configuration
public class MvcConfig extends WebMvcConfigurer {


    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/home").setViewName("home");
        registry.addViewController("/").setViewName("home");
        registry.addViewController("/hello").setViewName("hello");
        registry.addViewController("/login").setViewName("login");
    }


}
```

Example : *Interceptors*

```
@SpringBootApplication
public class MyApplication {

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

    @Bean
    public WebMvcConfigurer adapter() {
        return new WebMvcConfigurer() {
            @Override
            public void addInterceptors(InterceptorRegistry registry) {
                System.out.println("Adding interceptors");
registry.addInterceptor(new MyInterceptor().addPathPatterns("/**");

                super.addInterceptors(registry);
            }
        };
    }
}
```

HTTP message converter

SpringBoot provides default converters to process JSON, XML, String data into UTF-8

You can add own converters processing a particular type of media.

```
@Configuration
public class MyConfiguration {

    @Bean
    public HttpMessageConverters customConverters() {
        HttpMessageConverter<?> additional = ...
        HttpMessageConverter<?> another = ...
        return new HttpMessageConverters(additional, another);
    }
}
```

Static content

By default, SpringBoot serves static content from the ***/static*** directory (or */public* or */resources* or */META-INF/resources*) in the classpath

- It then uses *ResourceHttpRequestHandler* from SpringMVC

Locations can be changed by property: `spring.resources.staticLocations`

Webjars

- Client libraries (ex: jQuery, bootstrap, ...) can be packaged in jars : ***webjars***
  - Webjars allow dependency management by Maven

- Spring is able to serve Webjars resources present in an archive located in /webjars/**

Example

```
<dependency>
<groupId>org.webjars</groupId>
<artifactId>bootstrap</artifactId>
<version>3.3.7-1</version></dependency>

…
href = /webjars/bootstrap/3.3.7-1/css/bootstrap.min.css
```

View and templating technologies

- Spring MVC can generate dynamic html using basic templating technology.
- Spring Boot allows self-configuration of
  - FreeMarker
  - Groovy
  - Thymeleaf
  - Mustache
- The templates are then taken from the location ***src/main/resources/templates***

Errors handling

- Spring Boot associates ***/error*** with the global application error page
  - Exceptions thrown during request redirect to these error page by default
  - A default behavior in REST or in Web allows to visualize the cause of the error
- To override the default behaviour
  - Implement ***ErrorController*** and save it as Bean
  - Add a bean of type ***ErrorAttributes*** that overrides the content of the error page
  - Add a class annotated by ***@ControllerAdvice*** to customize the returned content
  - Design one or several custom pages

Exemple *@ControllerAdvice*

```
@ControllerAdvice
public class NotFoundAdvice extends ResponseEntityExceptionHandler {

  @ExceptionHandler(value = {MemberNotFound.class, DocumentNotFoundException.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);
  }
}
```

Error page

- To override the default view of the /error page, simply construct an error view.
  - Typically, if using Thymeleaf, write an ***error.html*** view in the template directory

Custom error pages

If you want to display error pages according to the HTTP return code. Just add static or dynamic pages in the /error directory

```
src/
 +- main/
     +- java/
     |   + <source code>
     +- resources/
         +- public/
             +- error/
             |   +- 404.html
             |   +- 5xx.ftl
```

Reactive Spring

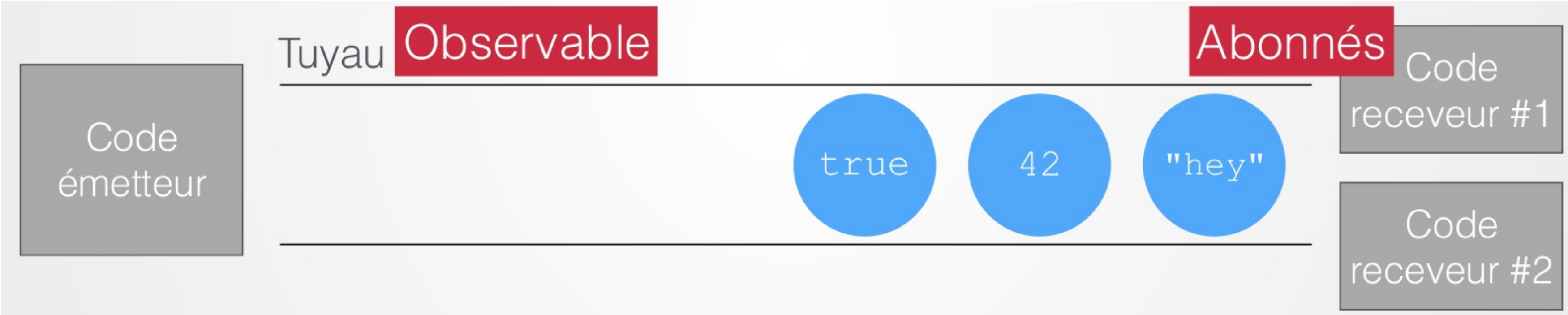
Reactive programmation
Spring Reactor
Spring Data Reactive
Spring Webflux

Reactive model

Consists of building your code from data streams: ***stream***.

Some parts of the code emit data : ***Observables*** or ***Publishers***

Others react from Event : ***Subscribers*** :



Observables/Publishers 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 subscriber.

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

Pattern Observable and *ReactiveX*

- Reactive programming is based on the ***Observable*** pattern which is a combination of *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)

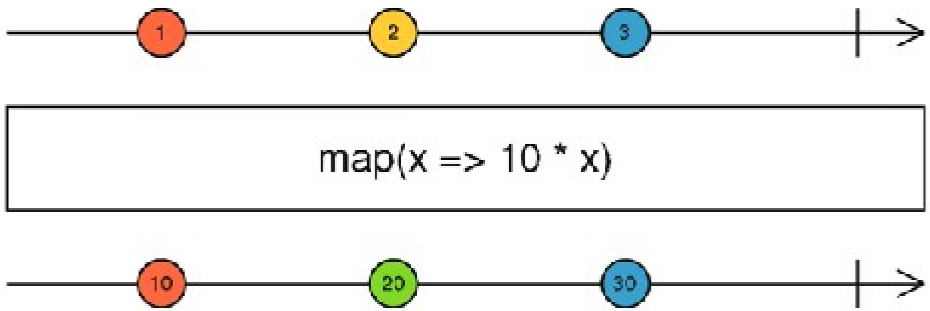
Combine transformations

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

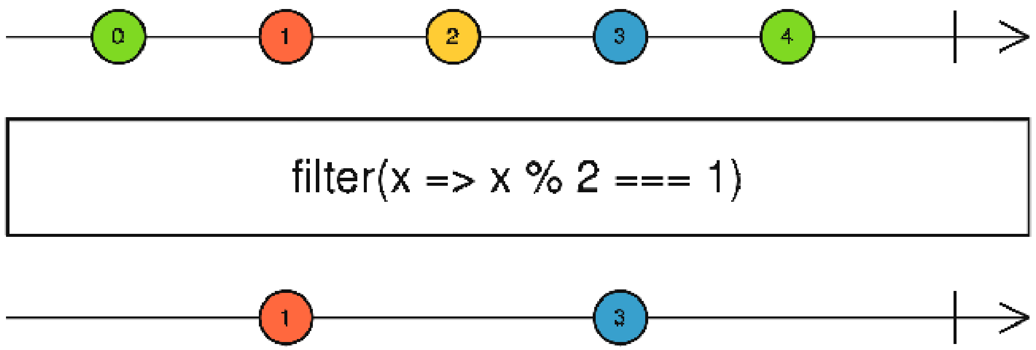
The operators to apply to the data are declared once and for all.

Marble diagrams

- Documentation for operators use ***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

Reactive Streams Interfaces

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

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

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

```java
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.

Reactive Stream helps establish the mechanism and set the frame rate limits via the method:

void request(long n) of 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 the highest level ***Mono*** and ***Flux*** types representing stream of events
  - It offers a set of operators aligned with *ReactiveX*.
  - It is an implementation of *Reactive Streams*

Reactive Spring

Programmation réactive

Spring Reactor

Spring Data Reactive

Spring Webflux

Maven Dependency

```
<dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-core</artifactId>
    <version>${version}</version>
</dependency>
```

2 Types

Reactor mainly offers 2 Java types:

- ***Mono*** : Stream of 0..1 element
- ***Flux*** : Stream 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<T>*** represents an asynchronous sequence of 0 to N events, optionally terminated by an end signal or an error.

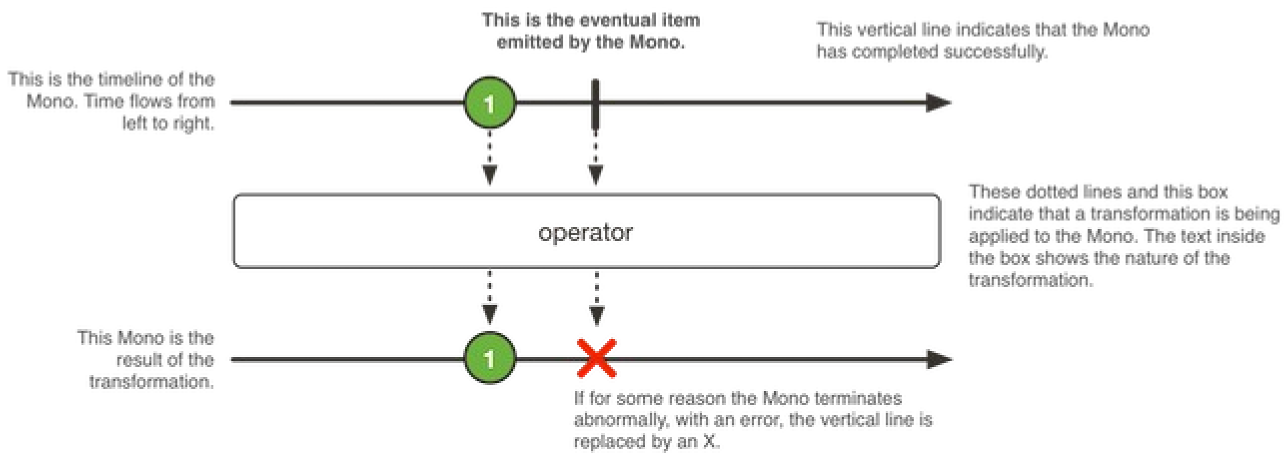
Events are translated to method calls on subscribers:

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

Mono

Mono<T> represents a sequence of 0 to 1 events, optionally terminated by an end signal or an error

Mono offers a subset of 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(bookCollecion.stream());
```

Subscription

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

Generally, lambda-expressions are used

```
subscribe();  // Trigger the flux
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 the emission of n 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); // Trigger the emission of all events
    }

    @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 (nouvel objet), *cast* (chgt de type), *index*(Tuple avec ajout d'un indice)

1 to N :
  flatMap + one factory method, *handle*

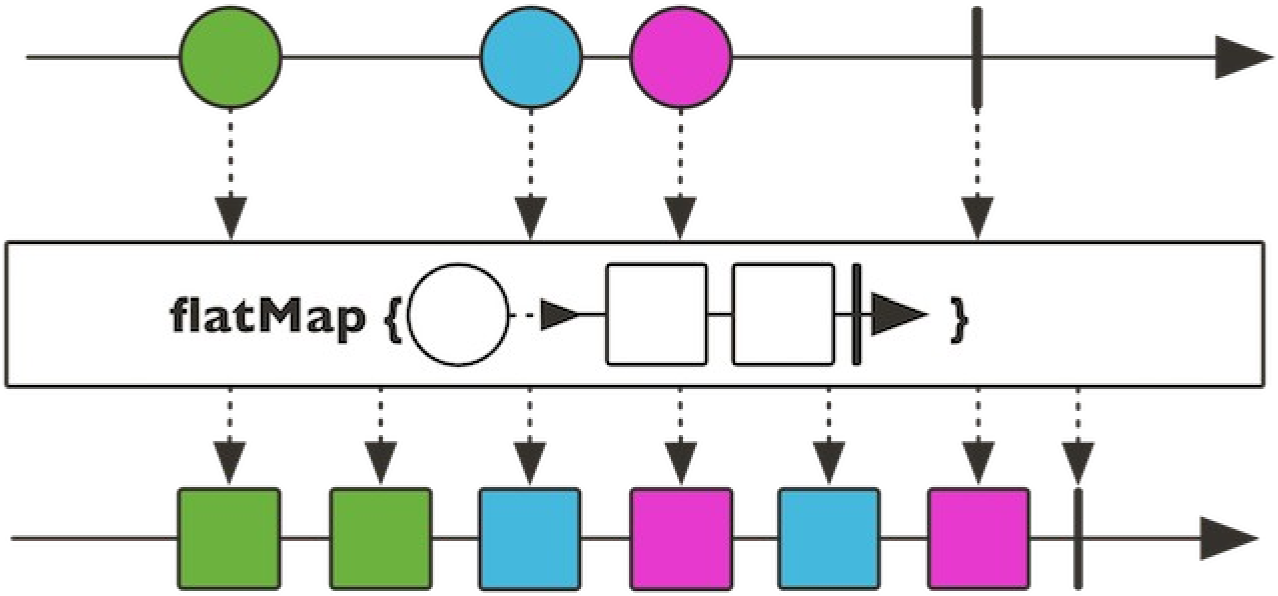
Add elements to sequence :
  startsWith, *endWith*

Agregate:
  collectList, *collectMap*, *count*, *reduce*, *scan*,

Agregate to boolean :
  all, *any*, *hasElements*, *hasElement*

Combine several Publisherq :
  concat, *merge*, *zip*

flatMap



Filters

Filter with an arbitrary function :
filter

On the type :
ofType

Ignore all :
ignoreElements

Ignore duplicates:
distinct

Only a subset :
take, **_takeLast_**, **_elementAt_**

Skip some :
skip(Long | Duration), skipWhile

Temporal operators

Associate the event with a timestamp :
elapsed, timestamp

Sequence interrupted if too much delay between 2 events:
timeout

Sequence at regular intervals:
interval

Add delays
Mono.delay, delayElements, delaySubscription

Retrurn to synchronous mode

On Flux :

- *bockFirst* : block until the first element. ...can precise a timeout
- *blockLast* : block until the last element (or null if empty): can precise a timeout:
- *toIterable*, *toStream* : synchronously switch to classic Java classes

On Mono

- *block* : optionaly with a timeout
- *toFuture* : a CompletableFuture<T>:

Alternate Publisher (Non blocking)

- *switchIfEmpty* : Provide a alternate Publisher if the stream is empty
- *then* : Return a Mono when the flux ends

Reactive Spring

Programmation réactive
Spring Reactor
Spring Data Reactive
Spring Webflux

Introduction

- Reactive programming also invites itself in SpringData
- Attention, this does not concern JPA which remain blocking APIs
- Are supported:
  - MongoDB
  - Cassandra
  - Redis
  - JDBC avec R2DBC

Reactive access to persistent data

- Calls are asynchronous, non-blocking, event-driven
- Data is handled as streams
- It needs :
  - Spring Reactor
  - Spring Framework 5
  - Spring Data 2.0
  - A reactive driver
  - Optionally Spring Boot (2.x+)

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*.

Benefits of Spring Data Reactive

Functionality remain close to Spring Data concepts:

- Reactive Templates classes
- Reactive Repository interfaces
- Data return are wrapped with Flux or Mono

Reactive Template

The Template classes 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)
..
```

Queries

queries can be inferred from method 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);
}
```

Example dependencies for *MongoDB* with *SpringBoot*

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.0.0.BUILD-SNAPSHOT</version>
</parent>
< !-- Ramène en particulier : spring-data-mongodb et reactor-core -->
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>

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

Reactive Spring

Programmation réactive
Spring Reactor
Spring Data Reactive
Spring Webflux

Motivation

2 main motivations for Spring Webflux:

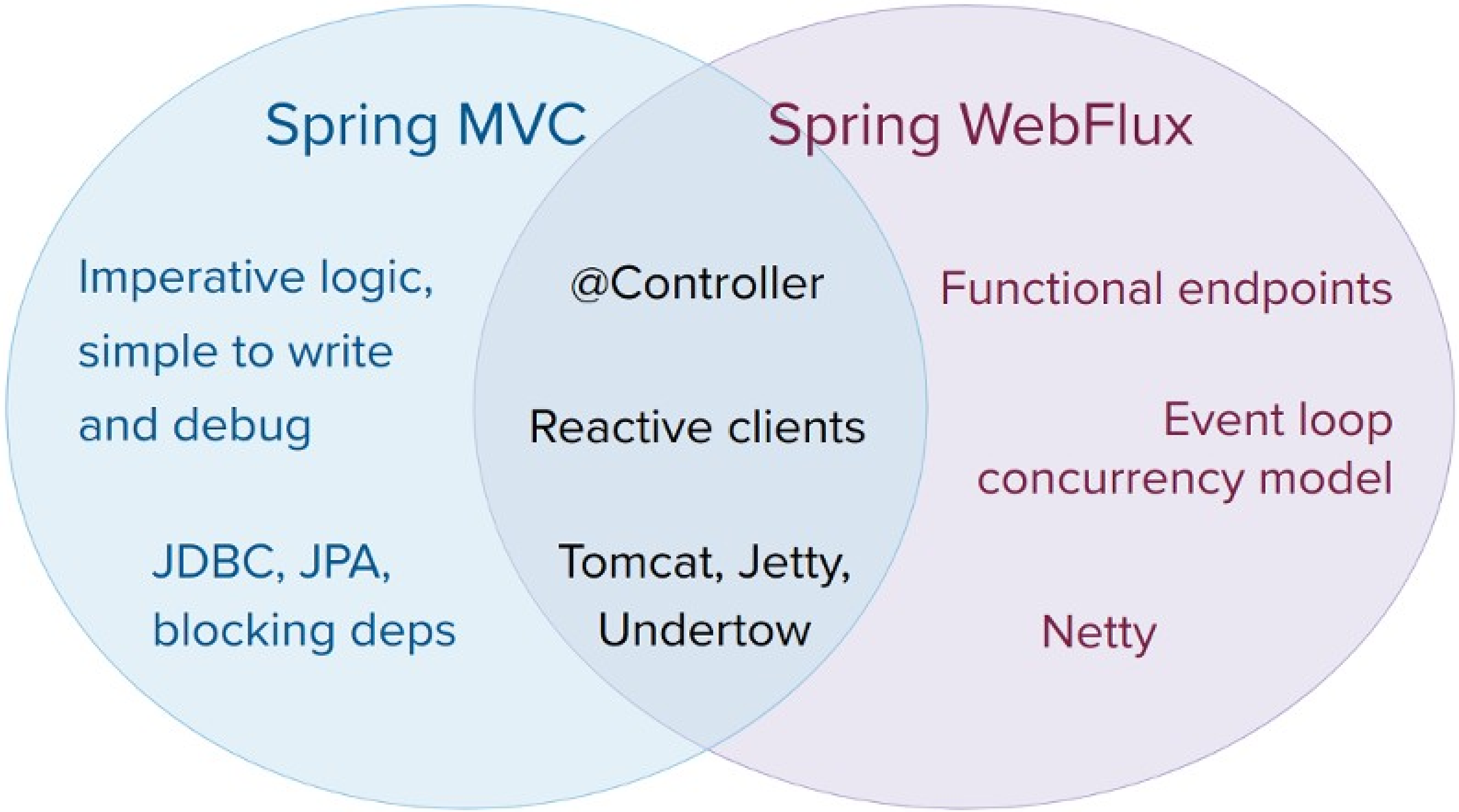
- The need for a non-blocking stack allowing to manage concurrency with few threads and to scale with less CPU/memory resources
- Functional programming

Programming models

2 programming modes are available :

- **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 applications 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

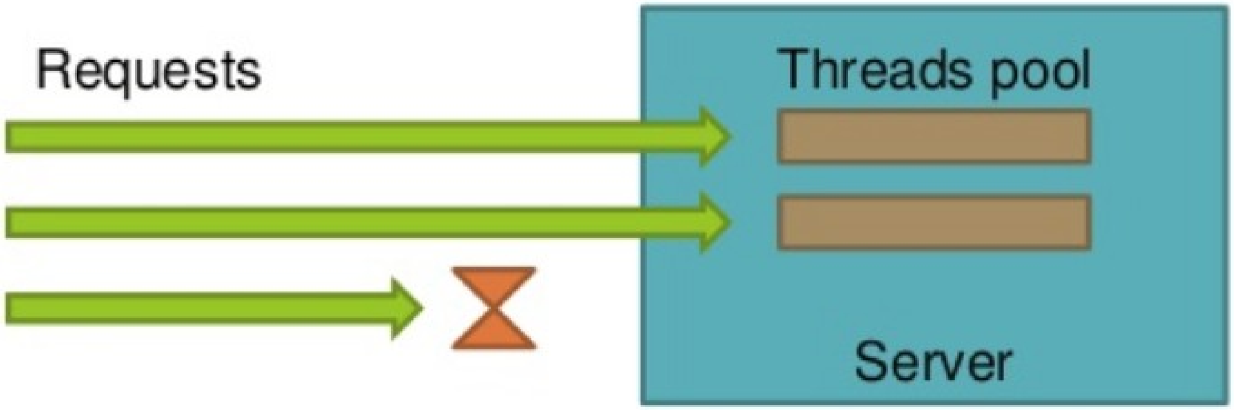
- Tomcat, Jetty, and Servlet containers 3.1+,
- non-Servlet environments like Netty or Undertow

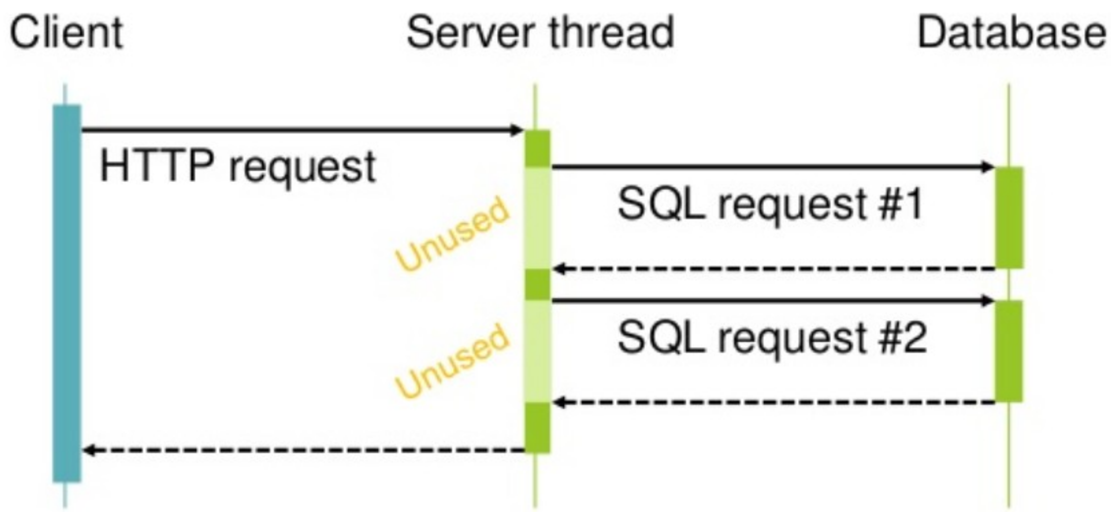
With SpringBoot, the default configuration starts an embedded Netty server

Performance and Scaling

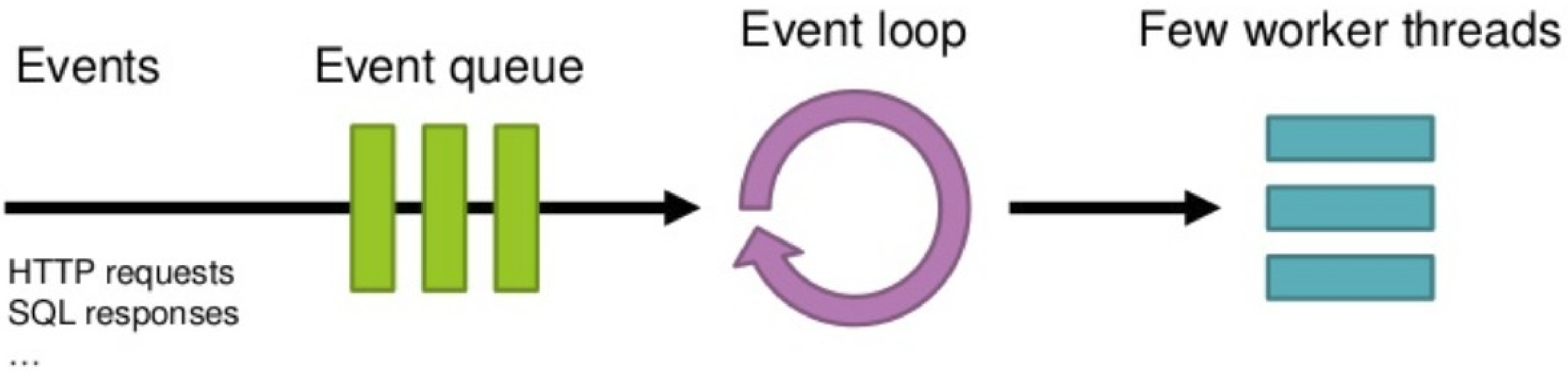
- The reactive and non-blocking model does not bring any particular gain in terms of response time. (there is more to do and it may even increase processing time)
- The expected benefit is the ability to **scale** with a small number of fixed threads and less memory.
- This makes applications more resistant 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



Thread models

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 bocking I/O (like JPA , it is required 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<Person> create(@RequestBody Person person) {
  return this.repository.save(person);
}
@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 + Meta-data
  - ***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 responsive template

ServerRequest and *ServerResponse*

ServerRequest et ***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);
  ```

Handling requests with *HandlerFunction*

HandlerFunction is a funnction which takas as input a *ServerRequest* and return 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 *HandlerFunction* 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);
```

Filters with *HandlerFilterFunction*

Routes controlled by a routing function can be filtered:

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

HandlerFilterFunction is a function which takes a *ServerRequest* and a *HandlerFunction* and returns a *ServerResponse*.

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

Exemple : *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();
        }
  });
```

Personnalisation de la configuration

La configuration par défaut peut être surchargée par différent moyens

- Les propriétés qui modifient les valeurs par défaut des beans techniques via :
  - Des **variables d'environnement**
  - Des **arguments de la ligne de commande**
  - Des **fichiers de configuration externe** (*.properties* ou *.yml*).
    Différents fichiers peuvent être activés en fonction de profils
- Avec des classes de configuration redéfinissant les beans par défaut
- En utilisant des **classes spécifiques du framework** (exemple classes **Configurer*)

Autres apports de SpringBoot

En dehors de l'auto-configuration, SpringBoot offre d'autres bénéfices pour les développeurs :

- Simplification de la gestion des dépendances vers les librairies OpenSource via les **starters**
- Assistant de création de projet avec **SpringInitializer**
- Point central de la configuration des propriétés avec ***application.properties/yml***
- Plugins pour les **IDEs** (SpringTools suite ou autre)
- Plugins pour les **outils de build** (Gradle et Maven)

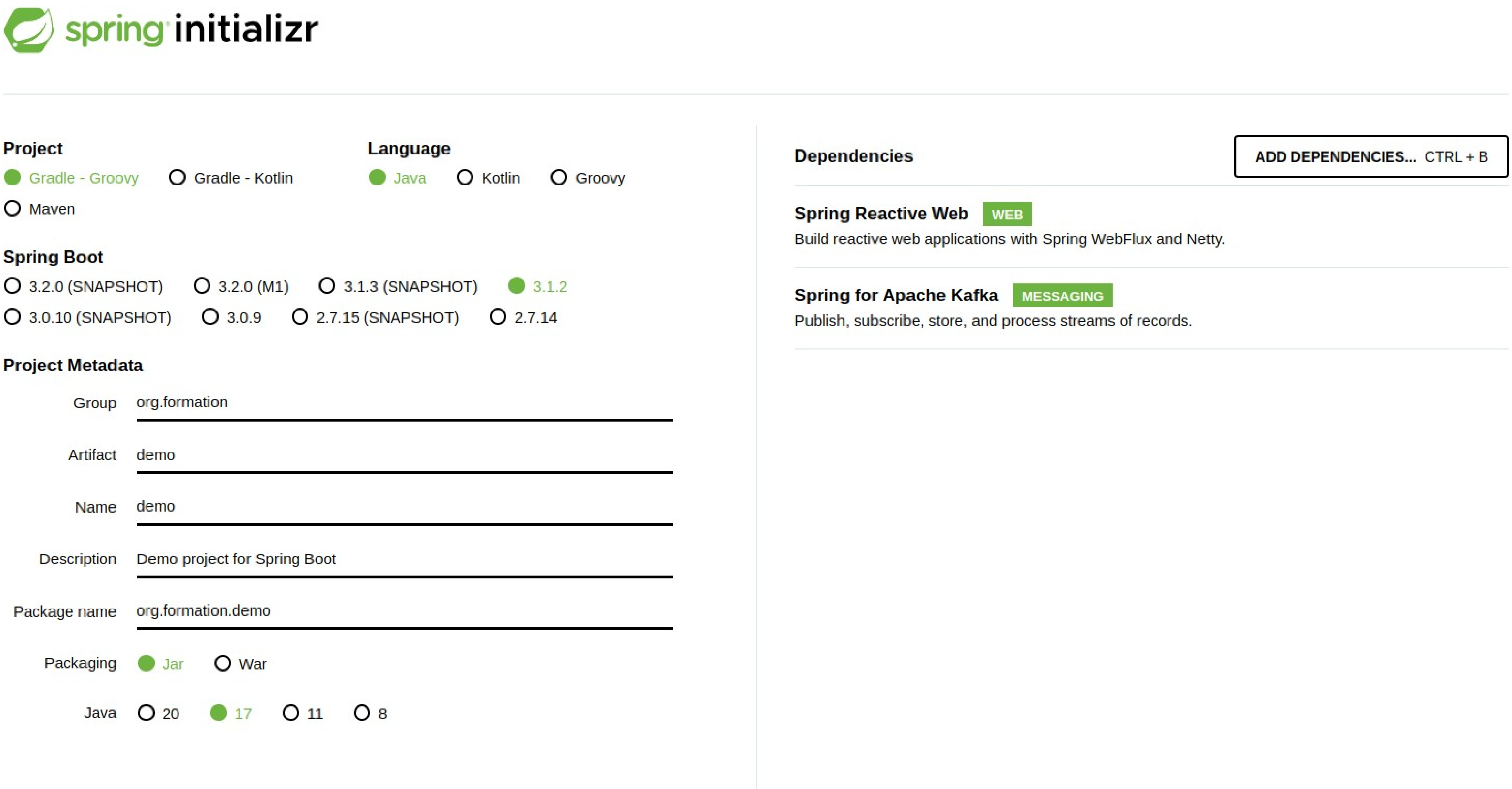
Gestion des dépendances

La gestion des dépendances est simplifiée grâce aux **starters**.

Ce sont des groupes de dépendances permettant d'intégrer une technologie ou apportant une fonctionnalité qui sont déclarés dans le fichier de build

=> Pour les développeurs, ce mécanisme simplifie à l'extrême la gestion des versions des dépendances.
Plus qu'un seul numéro de version à gérer : Celui de SpringBoot

https://start.spring.io/



Fichiers générés par l'assistant

– *.gitignore, Help.md*

– Scripts de build (*mvnw* ou *gradlew*)

– Classe de démarrage de SpringBoot (*src/main/java*)

– Classe de test de la configuration (*src/test/java*)

– Fichier de configuration des propriétés (*src/main/resources*)

Exemple Gradle

```
plugins {
  id 'java'
  id 'org.springframework.boot' version '3.1.2'
  id 'io.spring.dependency-management' version '1.1.2'
}

group = 'org.formation'
version = '0.0.1-SNAPSHOT'

java {
  sourceCompatibility = '17'
}

repositories { mavenCentral() }

dependencies {
  implementation 'org.springframework.boot:spring-boot-starter-webflux'
  implementation 'org.springframework.kafka:spring-kafka'
  testImplementation 'org.springframework.boot:spring-boot-starter-test'
  testImplementation 'io.projectreactor:reactor-test'
  testImplementation 'org.springframework.kafka:spring-kafka-test'
}

tasks.named('test') {   useJUnitPlatform()
```

Plug-in Maven/Gradle de SpringBoot

L’initializer crée des scripts (***mvnw*** ou ***gradlew***) pour les environnements Linux et Windows.

– Ce sont des wrappers de l’outil de build garantissant que tous les développeurs utilisent la même version de l’outil de build.

La commande la + importante dans un contexte Maven :

./mvnw clean package

=> Génère un *fat-jar*

L’application peut alors être démarrée en ligne de commande par :

java -jar target/myAppli.jar

Avec gradle :

```
gradle build
java -jar build/libs/mymodule-0.0.1-SNAPSHOT.jar
```

Autres tâches de build

Packager une image OCI :
spring-boot:build-image

Exécuter l'application
spring-boot:run

Exécuter avec le classpath de tes
spring-boot:***test-run***

Construire une image native
package -***Pnative***

Exécuter des tests d'intégration
spring-boot:***start***
spring-boot:***stop***

Renseigner le endpoint /*actuator/info* avec les informations de build
spring-boot:***build-info***

Classe de démarrage

La classe de démarrage est annotée via ***@SpringBootApplication***, annotation qui englobe :

- ***@Configuration :*** Permet de définir des méthodes *@Bean*
- ***@ComponentScan*** : Scan des annotations dans les sous-packages
- ***@EnableAutoConfiguration*** : Activation du mécanisme d'auto-configuration de SpringBoot

```
@SpringBootApplication
public class DemoApplication {

public static void main(String[] args) {
  // Création du contexte Spring
  SpringApplication.run(DemoApplication.class, args);
}


}
```

Classe de test

- La classe de test est annotée via ***@SpringBootTest*** :
  - Permet de créer le contexte Spring avant l'exécution du test junit5

```
@SpringBootTest
class DemoApplicationTests {

@Test
void contextLoads() {
  // Si le test passe, la configuration SpringBoot est OK
}


}
```

SpringBoot

L’auto-configuration
Starters SpringBoot
Structure projet et principales annotations
Propriétés de configuration

Starters les + importants

Web

***-web** : Application web ou API REST. Framework *Spring MVC*

***-reactive-web** : Application web ou API REST en mode réactif. Framework *Spring WebFlux*

Cœurs :

***-logging :** Utilisation de logback (Tjs présent)

***-test** : Test avec Junit, Hamcrest et Mockito (Tjs présent)

Starters développement

-devtools : Fonctionnalités pour le développement

-lombok : Simplification du code Java

-configuration-processor : Complétion des propriétés de configuration applicatives disponibles dans l'IDE

-docker-compose[1] : Support pour démarrer les services de support via docker-compose (BD, Kafka, etc..)

-graalvm-native[1] : Support pour construire des images natives

-modulith[1] : Support pour construire des applications monolithes modulaires

1. Depuis Spring 3.x

Sécurité

- ***-security** : *Spring Security*, sécurisation des URLs et des services métier
- ***-oauth2-client** : Pour obtenir un jeton *oAuth* d'un serveur d'autorisation
- ***-oauth2-resource-serve**r : Sécurisation des URLs via *oAuth*
- ***-ldap** : Intégration LDAP
- ***-okta** : Intégration avec le serveur d'autorisation Okta

Starters SQL

jdbc : JDBC avec pool de connexions Tomcat

Spring Data JPA: Spring Data avec Hibernate et JPA

Spring Data JDBC : Spring Data avec jdbc

Spring Data R2DBC : Spring Data avec *jdbc reactif*

MyBatis : Framework MyBatis

LiquiBase Migration : Migration de schéma avec Liquibase

Flyway Migration : Migration de schéma avec Flyway

JOOQ Access Layer : API fluent pour construire des requêtes SQL

*-***<drivers>*** : Accès aux driver JDBC (MySQL, Postgres, H2, HyperSonic, DB2)

Starters NOSQL

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

-data-neo4j : Base de données orienté graphe de Neo4j

-data-couchbase *-data-reactive-couchbase : Base NoSQL CouchBase

-data-redis *-data-reactive-redis : Base NoSQL Redis

-data-geode : Stockage de données via Geode

-data-elasticsearch : Base documentaire indexée ElasticSearch

-data-solr: Base indexée SolR

-data-mongodb *-data-reactive-mongodb : Base NoSQL MongoDB

Messaging

-integration: Spring Integration (Abstraction de + haut niveau pour implémenter des patterns d'intégration de façon déclaratif)

-kafka: Intégration avec Apache Kafka

-kafka-stream: Intégration avec l'API KafkaStream

-rabbitmq: Intégration avec Rabbit MQ

-activemq5 : ActiveMQ avec JMS

-artemis : ApacheMQ avec Artemis

*-***pulsar, *-reactive-pulsar*** : Messagerie PULSAR

-websocket : Servlet avec STOMP et SockJS

-rsocket : SpringMessaging et Netty

-camel : Intégration avec Apache Camel

-solacePubSub : Intégration avec Solace

Autres Starters Web

Moteur de templates HTML

****-thymeleaf :*** *Spring MVC* avec des vues *ThymeLeaf*

****-mustache*** *: Spring MVC* avec *Mustache*

****-groovy-templates*** *: Spring MVC avec* gabarits *Groovy*

****-freemarker****: Spring MVC* avec *freemarker*

Autres

****-graphql*** : *API GraphQL*

****-rest-repository, restrepository-explorer, *-hateoas*** : Génération API Rest à partir des repositories de Spring Data

****-jersey*** : API Restful avec JAX-RS et Jersey

****-webservices***: Services SOAP

****-vaadin, *-hila*** : Framework pour applis web

Autres Starters

I/O

***-batch** : Gestion de batchs

***-mail** : Envois de mails

***-cache**: Support pour un cache

***-quartz** : Intégration avec Scheduler

***-shell** : Support pour commandes en ligne

Ops

***-actuator** : Points de surveillance REST ou JMX

***-spring-boot-admin (client et serveur)** : UI au dessus d'actuator

*-***sentry*** : Intégration sentry (monitoring performance)

Spring Cloud

Services cloud : Facilité de déploiement

Amazon, Google Cloud, Azure, Cloud Foundry, Alibaba

Architecture Micro-services

Services de discovery, de configuration externalisée, de répartition de charge, de gateway, de circuit breaker

Spring Cloud Contract : Génération de tests et mock servers

Service de monitoring, de tracing, etc ...

Observabilité

Depuis la version 3.x, SpringBoot s’appuie fortement sur ***Micrometer***.

– Des starters permettent de publier les métriques *micrometer* vers des système de visualisation :
  DataDog, Dynatrace, Influx, Graphite, New Relic, Prometheus, Wavefront

– D’autres starters permettent la tracabilité des requêtes dans les architecture micro-services :
  Brave et Zipkin