Spring & Java

Plan de formation

- Syntaxe java : date et collections
- Spring boot
- Api REST avec Spring
- Persistance avec JDBC
- Persistance avec JPA et transaction
- Spring actuators
- Spring security pour les API
- Webflux
- Introduction aux modules Java 9

Tout au long de la formation : discussion ouverte sur la gestion de la mémoire, Kotlin et dont ce dont vous vous parlez

Java, Date, interface et Collection

Interface

- Définit les méthodes à implementer ou pas…
- Ne peut pas être instanciée, ni être **final.**
- Déclarée grâce au mot-clé **interface.**
- Peut contenir des membres **static** et **final.**
- Une classe pour implementer plusieurs interface.
- Une interface peut étendre plusieurs interfaces.
- Par défaut, tous les membres sont **public et static.**

Méthodes default

- Méthode par défaut
- Mot clé **default** dans la signature et corps de la méthode à définir.
- Problème de l'héritage en losange
  - Signer l'appel de la méthode par l'interface correspondante.
- Factorisation du corps de la méthode

```
public interface IsWarmBlooded { boolean hasScales();
    public default double getTemperature() {
        return 10.0; }
}
```

Objets autoclosable – Try avec ressources

- On peut placer des ressources dans le try
  - try(Oneone=newOne();Twotwo=newTwo()){...}
- Interface permet de fermer, non de collecter (GC), automatiquement un objet après son utilisation
  - Méthode close()
  - Une fois fermée, la méthode close ne fait plus rien (idempotent)
  - Gestion des Exceptions
  - Les exceptions du close sont « suppressed » dans le try où est clos l'objet
- Closeable
  - Fermée à la main
  - Aucune garantie de fermeture
  - Gestion des IOException

Date

- Date actuelle
  - **LocalDateTime** currentTime = **LocalDateTime.now();**
    - Current DateTime: 2014-12-09T11:00:45.457
  - **LocalDate** date1 = **currentTime.toLocalDate();**
    - date1: 2014-12-09
  - Month month = currentTime.getMonth();
    - Month: DECEMBERday
  - int day = currentTime.getDayOfMonth();
  - **LocalTime** date1 = **LocalTime.now();**

Date précise

- LocalDateTime date2 = **currentTime.withDayOfMonth(10).withYear(2012);**
  - date2: 2012-12-10T11:00:45.457
- LocalDate date3 = **LocalDate.of(2014, Month.DECEMBER, 12)**
  - date3: 2014-12-12
  - Pour les mois, les chiffres sont possible en langage humain

Temps précis

- LocalTime time1 = LocalTime.of(6, 15);
- LocalTime time2 = LocalTime.of(6, 15, 30);
- LocalTime time3 = LocalTime.of(6, 15, 30, 200);
- Ordre : heure, minute, secondes, nanosecondes

Date et heure données

- LocalDateTime dateTime1 = LocalDateTime.of(2015, Month.JANUARY, 20, 6, 15, 30);
- LocalDateTime dateTime2 = LocalDateTime.of(date1, time1);
- Signature des méthodes :
  - public static LocalDateTime of(int year, int month, int dayOfMonth, int hour, int minute)
  - public static LocalDateTime of(int year, int month, int dayOfMonth, int hour, int minute, int second)
  - public static LocalDateTime of(int year, int month, int dayOfMonth, int hour, int minute, int second,int nanos)
  - public static LocalDateTime of(int year, Month month, int dayOfMonth, int hour, int minute)
  - public static LocalDateTime of(int year, Month month, int dayOfMonth, int hour, int minute, int second)
  - public static LocalDateTime of(int year, Month month, int dayOfMonth, int hour, int minute, int second, int nanos)
  - public static LocalDateTime of(LocalDate date, LocalTime)

Méthodes

	Avec LocalDate?	Avec LocalTime?	Avec LocalDateTime?
plusYears/minusYears	X		X
plusMonths/minusMonths	X		X
plusWeeks/minusWeeks	X		X
plusDays/minusDays	X		X
plusHours/minusHours		X	X
plusMinutes/minusMinutes		X	X
plusSeconds/minusSeconds		X	X
plusNanos/minusNanos		X	X

Exemple :

```
LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
LocalTime time = LocalTime.of(5, 15);
LocalDateTime dateTime = LocalDateTime.of(date2, time)
      .minusDays(1).minusHours(10).minusSeconds(30);
```

Formatage : DateTimeFormatter

```
LocalDateTime now = LocalDateTime.now();
System.out.println("Before : " + now);
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
String formatDateTime = now.format(formatter);
System.out.println("After : " + formatDateTime);
LocalDateTime formatDateTime = LocalDateTime.parse(now, formatter);
System.out.println("After : " + formatDateTime.format(formatter));
```

- Before : 2016-11-09T11:44:44.797
- After : 2016-11-09 11:44:44
- After : 2016-11-09T10:30

Formatters accessibles

ofLocalizedDate(dateStyle)	Formatter with date style from the locale	'2011-12-03'
ofLocalizedTime(timeStyle)	Formatter with time style from the locale	'10:15:30'
ofLocalizedDateTime(dateTimeStyle)	Formatter with a style for date and time from the locale	'3 Jun 2008 11:05:30'
ofLocalizedDateTime(dateStyle,timeStyle)	Formatter with date and time styles from the locale	'3 Jun 2008 11:05'
BASIC_ISO_DATE	Basic ISO date	'20111203'
ISO_LOCAL_DATE	ISO Local Date	'2011-12-03'
ISO_OFFSET_DATE	ISO Date with offset	'2011-12-03+01:00'
ISO_DATE	ISO Date with or without offset	'2011-12-03+01:00'; '2011-12-03'
ISO_LOCAL_TIME	Time without offset	'10:15:30'
ISO_OFFSET_TIME	Time with offset	'10:15:30+01:00'

Formatters accessibles

ISO_TIME	Time with or without offset	'10:15:30+01:00'; '10:15:30'
ISO_LOCAL_DATE_TIME	ISO Local Date and Time	'2011-12-03T10:15:30'
ISO_OFFSET_DATE_TIME	Date Time with Offset	2011-12-03T10:15:30+01:00'
ISO_ZONED_DATE_TIME	Zoned Date Time	'2011-12-03T10:15:30+01:00[Europe/Paris]'
ISO_DATE_TIME	Date and time with ZoneId	'2011-12-03T10:15:30+01:00[Europe/Paris]'
ISO_ORDINAL_DATE	Year and day of year	'2012-337'
ISO_WEEK_DATE	Year and Week	2012-W48-6'
ISO_INSTANT	Date and Time of an Instant	'2011-12-03T10:15:30Z'
RFC_1123_DATE_TIME	RFC 1123 / RFC 822	'Tue, 3 Jun 2008 11:05:30 GMT'

FormatStyle : dateStyle et timeStyle

- Enumération

FULL	Full text style, with the most detail.
LONG	Long text style, with lots of detail.
MEDIUM	Medium text style, with some detail.
SHORT	Short text style, typically numeric.

Parsing Pattern

- **MMMM** pour les mois
  - M pour 1,
  - MM pour 01,
  - MMM pour Jan
  - MMMM pour January.
- **dd** pour les jours.
  - d pour 1,
  - dd pour 01.
- **yyyy** pour les années, fonctionne avec yy ou yyyy.
- **hh** pour les heures, h sur un digit, hh sur deux digit.
- **mm** pour les minutes, m sur un digit, mm sur deux digits.

Parsing

- DateTimeFormatter f = DateTimeFormatter.ofPattern("MM dd yyyy");
- LocalDate date = LocalDate.parse("01 02 2015", f);
- LocalTime time = LocalTime.parse("11:22");
- System.out.println(date); // 2015-01-02
- System.out.println(time); // 11:22

ZoneDateTime

- Inclusion des fuseaux horaires

ZoneId zoneId = ZoneId.of("Europe/Paris"); And we can get a set of all zone ids:

ZonedDateTime zonedDateTime = ZonedDateTime.of(localDateTime, zoneId);

ZonedDateTime.parse("2015-05-03T10:15:30+01:00[Europe/Paris]");

Stockage et diffusion de dates

- Base de données SQL et API
- Uniquement la date :
  - LocalDate suffit
- Date et heure avec internationalisation :
  - **Timestamp**
  - **LocalDate**
    - Gérer le fuseau horaire à la main
  - **ZonedDateTime** à convertir en java.sql.timestamp

```
// Get current zonedDateTime
ZonedDateTime zonedDateTime = ZonedDateTime.now(ZoneOffset.UTC);
// Convert zonedDateTime to java.sql.Timestamp before saving to DB
Timestamp = Timestamp.from(zonedDateTime.toInstant());
// Get zonedDateTime from resultSet
Timestamp timestamp = (Timestamp) resultSet.getObject("created");
ZonedDateTime zonedDateTime = ZonedDateTime.ofInstant(ts.toInstant(), ZoneOffset.UTC))
```

Arrays

- Stockage de données mais de taille fixe
  - Declaration :

```
int[] myTab = new int[10];
int myTab[][] = new int[10][5];
int myTab[] = {1,2,3,4};
MyObject[] m = new MyObject[10];
```

  - Accès :

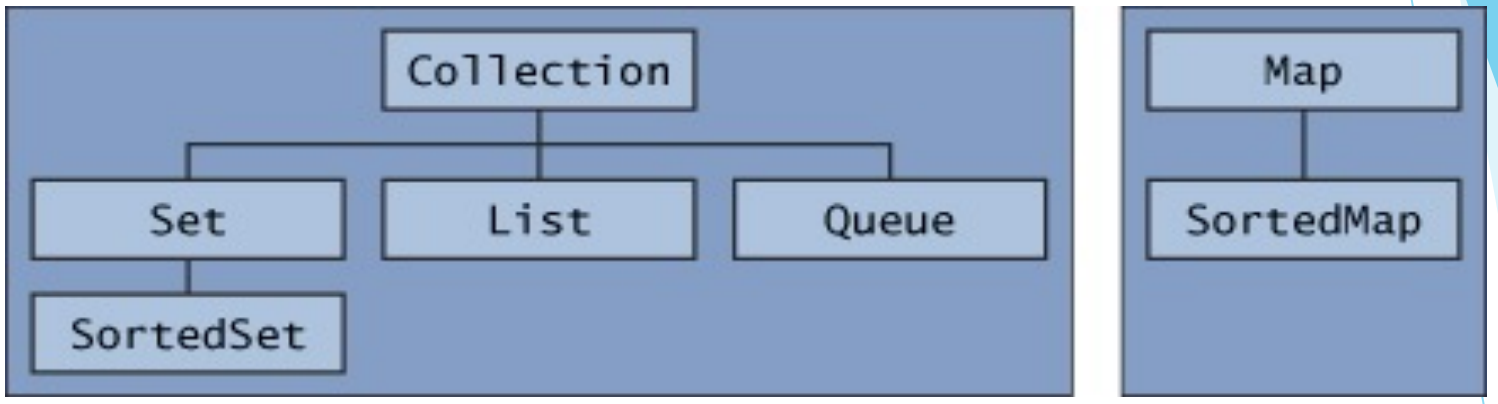
```
int firstElement = myTab[0];
```

Avantages des collections

- Taille dynamique
- Généricité
- Méthodes existants pour les opérations communes (insertion, tri, suppression…)

```
ArrayList<String> myString = new ArrayList<String>();

myString.add(" Hello ");
myString.add(" you ");
//a collection is never too small
myString.add("and you ");
```

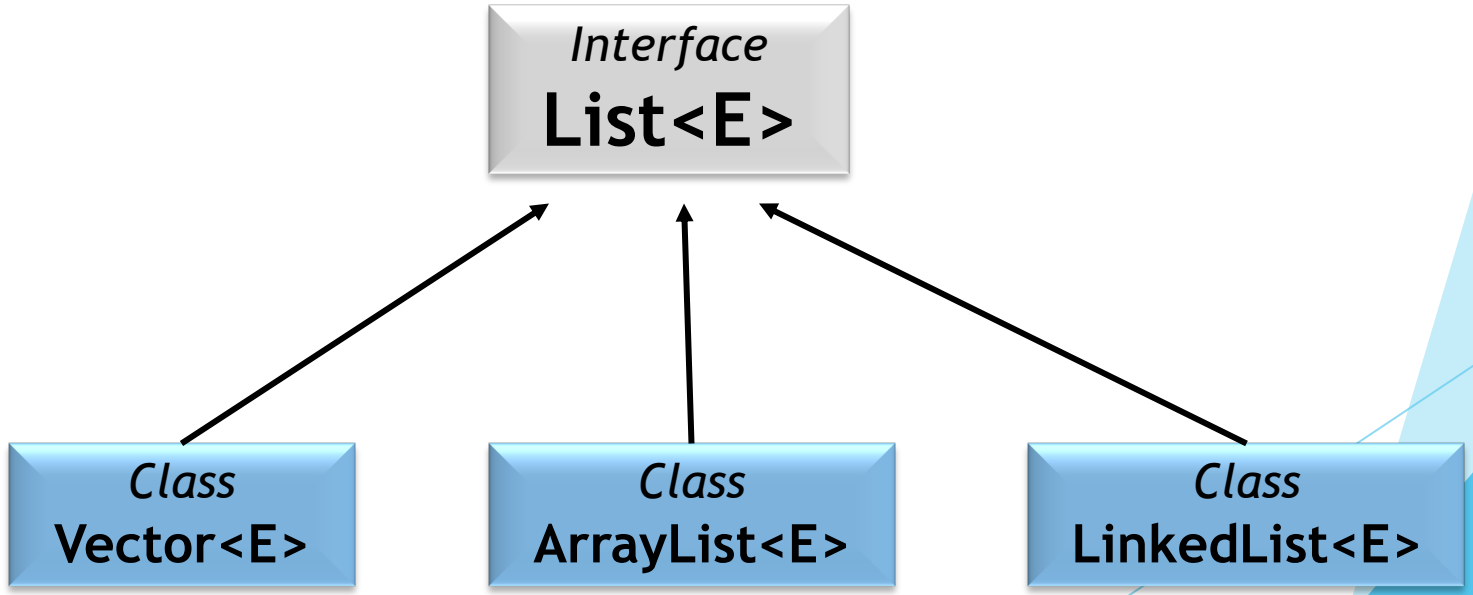
Les différentes collections



- Deus types distincts de collections.
- Toutes les interfaces utilisent la génricité.
- Chacune ont leurs avantages et inconvénients.
  - Ex: une Queue peut agir comme une pile (LIFO)
- A vous de choisir en fonction de vortre stratégie.

List interface

- Collection ordonnée.
- Accès aux éléments par leur indice.
- Peut contenir plusieurs fois le même élément.



Méthodes

- boolean add(E element)
- void add(int index, E element)
- boolean remove(Object object)
- E remove(int index)
- E set(int index, E newElement)
- boolean isEmpty()
- int size()
- void clear()
- boolean contains(Object object)

Exemple

```
List<String> myList = new ArrayList<String>();
myList.add("Monday");
myList.add("Tuesday");
myList.add("Wednesday");
myList.add("Thursday");
myList.add("Friday");
myList.add("Saturday");
myList.add("Sunday");

String day = myList.get(3);
System.out.println("The fourth day is " + day );
String removeDay = myList.remove(6);
System.out.println(“I removed " + removeDay );
```

Iterator presentation

- Un Iterator :
  - Est un objet générique.
  - Permet de parcourir une collection de manière linéaire.
  - Lit seulement en ordre ascendant.
  - Peut supprimer un élément : méthode **remove()**

```
boolean hasNext(); //Check if there is a next element
E next(); //Get the next element
void remove(); // Remove the current element
```

Iterator : Exemple

```
Collection<String> myCollection = new ArrayList<String>();
// Add elements to the collection.
myCollection.add("Remove me");
myCollection.add("Keep me");
Iterator<String> it = myCollection.iterator();
while (it.hasNext()) {
        String myElement = it.next();
        if (myElement.equals("Remove me")) {
                it.remove();
        } else {
                System.out.println(myElement);
        }
}
```

Foreach

- A la place des boucles for….

```
String [] tab = {"one", "two", "three", "four" };
for(int i = 0; i < 4; i++) {
    System.out.println(tab[i]);
}
```

```
String [] tab = {"one", "two", "three", "four" };
for(String s : tab) {
    System.out.println(s);
}
```

Foreach

- Depuis Java8 : utilisation de lambda possible

```
String [] tab = {"one", "two", "three", "four" };
tab.foreach(System.out.println)
```

Comparator pour trier une collection

- Utilisé pour trier les Collections.
- Classe implémentant l'interface **Comparator<E>**.
- Méthode à définir :

```
int compare(E e1, E e2);
```

- La valeur de retour est :
  - Positive si le premier élément est supérieur au second
  - 0 en cas d'égalité
  - Négative si le premier élément est inférieur au second.

Comparator: Exemple 1/2

- Implémentation

```
public class MyComparator implements Comparator<String> {
	public int compare(String s1, String s2){
		if(s1.length() == s2.length()) return 0;
		else if(s1.length() > s2.length()) return 1;
		else return -1;
	}
}
```

Comparator: Exemple 2/2

- Tri d'une List :

```
List<String> myList= new ArrayList<String>();
myList.add("John");
myList.add("Michael");
myList.add("Maria");
Collections.sort(myList, new MyComparator());
```

- Résultat : John, Maria, Michael

Comparable

- Autre moyen de trier les Collections.
- Classe implémentant l'interface **Comparable<E>**.
- Méthodes à définir :

```
int compareTo(E e);
```

- La valeur de retour est:
  - Négative si l'instance courante est inférieure au paramètre
  - 0 si égalité
  - Positivie si l'instance courante est supérieure au paramètre

Comparable: Exemple 1/2

- Implémentation :

```
public class User implements Comparable<User> {
        public String name;

        public User(String name) {
            this.name = name;
        }

        public int compareTo(User u2){
            //Reverse sort by name
            return - name.compareTo(u2.name);
        }
}
```

Comparable : Exemple 2/2

- Tri d'une Collection de Comparable :

```
…
ArrayList<User> mySet = new ArrayList<User>();
mySet.add(new User("John"));
mySet.add(new User("Michael"));
mySet.add(new User("Maria"));
…
Collections.sort(maList);
```

- Résultat : Michael, Maria, John.

Lambda

- Fonction anonyme pour :
  - déclaration d'une variable
  - assignation d'une variable
  - valeur de retour avec l'instruction return
    - initialisation d'un tableau
    - paramètre d'une méthode ou d'un constructeur ¤ corps d'une expression lambda
    - operateur ternaire ?:
    - cast
    - …

Syntaxe

- [final |@NotNull] paramètre(s) -> corps de la fonction
  - (paramètres) -> expression;
  - (paramètres) -> { traitements; }
  - Exemple :
  - () -> System.out.println("traitement");
  - Consumer<String> a cher = (param) -> System.out.println(param);
  - Consumer<String> afficher = String param -> System.out.println(param);
- Met en relation l'instruction avec la variable
- Le corps peut être une condition, une méthode à appeler, du code avec ou sans return...

Exemple

Exemple	Description
() -> 123 () -> { return 123 };	N'accepter aucun paramètre et renvoyer la valeur 123
x->x*2	Accepter un nombre et renvoyer son double
(x,y) ->x+y	Accepter deux nombres et renvoyer leur somme
(int x,int y) ->x+y	Accepter deux entiers et renvoyer leur somme
(String s) -> System.out.print(s)	Accepter une chaîne de caractères et l'afficher sur la sortie standard sans rien renvoyer

c->{int s=c.size();
c.clear(); return s;
}
	Renvoyer la taille d'une collection après avoir effacé tous ses éléments.
	Cela fonctionne aussi avec un objet qui possède une méthode size() renvoyant un entier et une méthode clear()

n->n%2!=0;	Renvoyer un booléen qui précise si la valeur numérique est impaire

Exemple

Exemple	Description
(char c) -> c ==' 'z;	Renvoyer un booléen qui précise si le caractère est 'z'
() -> { System.out.println("Hello World"); };	Afficher "Hello World" sur la sortie standard
(val1, val2) -> { return val1 >= val2; } (val1, val2) -> val1 >= val2;	Renvoyer un booléen qui précise si la première valeur est supérieure ou égale à la seconde
() ->{for(inti=0;i<10;i++) traiter(); }	Exécuter la méthode traiter() dix fois
p -> p.getSexe() == Sexe.HOMME && p.getAge() >= 7 && p.getAge() <= 77	Renvoyer un booléen pour un objet possédant une méthode getSexe() et getAge() qui vaut true si le sexe est masculin et l'age compris entre 7 et 77 ans

Références et lambda

Type	Référence de méthode	Expression lambda
Référence à une méthode statique	System.out::println Math::pow	x -> System.out.println(x) (x, y) -> Math.pow(x, y)
Référence à une méthode sur une instance	monObject::maMethode	x-> monObject.maMethode(x)
Référence à une méthode d'un objet arbitraire d'un type donné	String::compareToIgnoreCase	(x, y) -> x.compareToIgnoreCase (y)
Référence à un constructeur	MaClasse::new	() -> new MaClasse();

Exemple avec les interfaces

```
public class Calculatrice {
        @FunctionalInterface interface OperationEntiere {
                long effectuer(int a, int b);
        }
        public long calculer(int a, int b, OperationEntiere operation) {
                return operation.effectuer(a, b);
        }
        public static void main(String[] args) {
                Calculatrice calc = new Calculatrice();
                OperationEntiere addition = (a, b) -> a + b;
                OperationEntiere soustraction = (a, b) -> a - b;
                System.out.println(calc.calculer(10, 5, addition));
                System.out.println(calc.calculer(10, 5, soustraction));
        }
}
```

Exemple : tri d'une collection

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

public class TestComparator {
        public static void main(String[] args) {
                Personne p1 = new Personne("nom3","prenom3");
                Personne p2 = new Personne("nom1","prenom1");
                Personne p3 = new Personne("nom2","prenom2");
                List<Personne> personnes = new ArrayList(3)
                personnes.add(p1);
                personnes.add(p2);
                personnes.add(p3);
                Comparator<Personne> triParNom =
                (Personne pers1, Personne pers2) ->
                        { return p2.getNom().compareTo(p1.getNom()); };
        }
}
```

Exemple : tri d'une collection

```
public class ComparaisonPersonne {
        public int comparerParNom(Personne p1, Personne p2) {
                return p1.getNom().compareTo(p2.getNom()); }       public int
comparerParPrenom(Personne p1, Personne p2) {
                return p1.getPrenom().compareTo(p2.getPrenom()); } }

public class TestReferenceMethodeInstance {
        public static void main(String[] args) {
                Personne[] personnes = { new Personne("nom3", "Julien"), new
Personne("nom1", "Thierry"), new Personne("nom2", "Alain") };
                ComparaisonPersonne comparaisonPersonne = new
ComparaisonPersonne();
                Arrays.sort(personnes, comparaisonPersonne::comparerParNom);
                System.out.println(Arrays.deepToString(personnes));
                Arrays.sort(personnes, comparaisonPersonne::comparerParPrenom);
                System.out.println(Arrays.deepToString(personnes)); } }
```

Exemple : tri d'une collection

```
public class TestReferenceMethodeUnbound {

        public static void main(String[] args) {
                String[] fruits = {"Melon", "abricot", "fraise", "cerise",
"mytille"};
                Arrays.sort(fruits, String::compareToIgnoreCase);
                System.out.println(Arrays.deepToString(fruits));
        }
}
```

Exemple : tri d'une collection

```
public class TestReferenceMethodeUnbound {
        public static void main(String[] args) {
                String[] fruits = {"Melon", "abricot", "fraise", "cerise",
"mytillee"};
                Arrays.sort(fruits, (s1, s2) -> s1.compareToIgnoreCase(s2) );
                System.out.println(Arrays.deepToString(fruits));
        }
}
```

Streams

- Peu de chance à la certification
- Ouverture d’un flux de travail pour effectuer un ou plusieurs traitements sur un itérable
- Méthode .stream() sur un itérable

Méthodes des streams

Modifier and Type	Method and Description
boolean	allMatch(Predicate<? super T> predicate) Returns whether all elements of this stream match the provided predicate.
boolean	anyMatch(Predicate<? super T> predicate) Returns whether any elements of this stream match the provided predicate.
<R,A> R	collect(Collector<? super T,A,R> collector) Performs a mutable reduction operation on the elements of this stream using a Collector.
long	count() Returns the count of elements in this stream.
Stream<T>	distinct() Returns a stream consisting of the distinct elements (according to Object.equals(Object)) of this stream.
Stream<T>	filter(Predicate<? super T> predicate) Returns a stream consisting of the elements of this stream that match the given predicate.

Méthodes des streams

Modifier and Type	Method and Description
Optional<T>	findAny() Returns an Optional describing some element of the stream, or an empty Optional if the stream is empty.
Optional<T>	findFirst() Returns an Optional describing the first element of this stream, or an empty Optional if the stream is empty.
void	forEach(Consumer<? super T> action) Performs an action for each element of this stream.
Stream<T>	limit(long maxSize) Returns a stream consisting of the elements of this stream, truncated to be no longer than maxSize in length.
<R> Stream<R>	map(Function<? super T,? extends R> mapper) Returns a stream consisting of the results of applying the given function to the elements of this stream.
Optional<T>	max(Comparator<? super T> comparator) Returns the maximum element of this stream according to the provided Comparator.

Méthodes des streams

Modifier and Type	Method and Description
Optional<T>	min(Comparator<? super T> comparator) Returns the minimum element of this stream according to the provided Comparator.
boolean	noneMatch(Predicate<? super T> predicate) Returns whether no elements of this stream match the provided predicate.
Optional<T>	reduce(BinaryOperator<T> accumulator) Performs a reduction on the elements of this stream, using an associative accumulation function, and returns an Optional describing the reduced value, if any.
T	reduce(T identity, BinaryOperator<T> accumulator) Performs a reduction on the elements of this stream, using the provided identity value and an associative accumulation function, and returns the reduced value.

Méthodes des streams

Modifier and Type	Method and Description
Stream<T>	skip(long n) Returns a stream consisting of the remaining elements of this stream after discarding the first n elements of the stream.
Stream<T>	sorted() Returns a stream consisting of the elements of this stream, sorted according to natural order.
Stream<T>	sorted(Comparator<? super T> comparator) Returns a stream consisting of the elements of this stream, sorted according to the provided Comparator.
Object[]	toArray() Returns an array containing the elements of this stream.
<A> A[]	toArray(IntFunction<A[]> generator) Returns an array containing the elements of this stream, using the provided generator function to allocate the returned array, as well as any additional arrays that might be required for a partitioned execution or for resizing.

Exemple

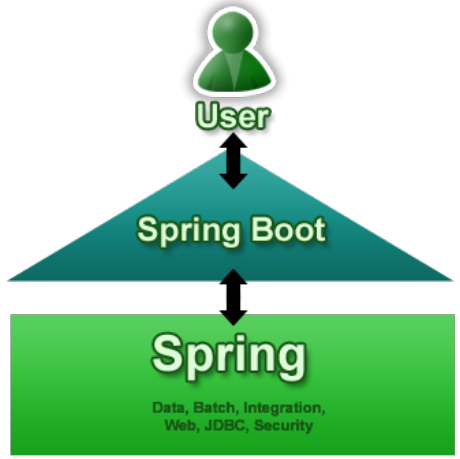
```
List<String> lines = Arrays.asList("spring", "node");
List<String> result = lines.stream()
        .filter(line -> !"spring".equals(line))
        .collect(Collectors.toList());
result.forEach(System.out::println);
```

- Des questions?

Spring boot

Introduction to Spring Boot

- Framework Spring basé sur les servlets qui
- Intègre un serveur Tomcat dans une application JSE
- => Simplification du développement
- => Architecture micro-services simplifiée également
- Avatanges principaux :
  - Autoconfiguration
  - Starter



graddle

```
plugins {
    id 'org.springframework.boot' version '2.5.2’
    id 'io.spring.dependency-management' version '1.0.11.RELEASE’
    id 'java’
}
group = 'com.example’
version = '0.0.1-SNAPSHOT’
sourceCompatibility = '1.8’
repositories {
     mavenCentral()
}
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-web’
    testImplementation('org.springframework.boot:spring-boot-starter-test') }
test { useJUnitPlatform() }
```

Main

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

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

Hello World API

```
package com.example.demo.controller;

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.ResponseBody;

@Controller
class ThisWillActuallyRun
{
    @RequestMapping("/")
    @ResponseBody
    String home() { return "Hello World!" }
}
```

Annotations

- L'annotation @Value est utilisable sur un attribut ou un paramètre pour un type primitif ou une chaîne de caractères. Elle donne la valeur par défaut à injecter.
- @Resource peut se substituer à l'annotation @Autowired sur les attributs et les méthodes *setter*. Le Spring Framework réalise une injection de dépendance basée sur le type attendu. Si l'annotation spécifie un nom grâce à son attribut name alors l'injection de dépendance se fait en cherchant un *bean* du même nom. Ex: @Resource(name = "tache")
- @PostConstruct s'utilise sur une méthode publique sans paramètre afin de signaler que cette méthode doit être appelée par le conteneur IoC après l'initialisation du *bean*. Il s'agit d'une alternative à la déclaration de la méthode d'initialisation.
- @PreDestroy s'utilise sur une méthode publique sans paramètre afin de signaler que cette méthode doit être appelée juste avant la fermeture du contexte d'application. Il s'agit d'une alternative à la déclaration de la méthode de destruction.

Application.properties

- Ce fichier est présent dans le chemin de classe (*classpath*) de l'application.
- Pour un projet géré avec Maven, cela signifie que le fichier application.properties se trouve dans src/main/resources.
- Ce fichier est utilisé pour paramétrer le comportement par défaut de l'application.
- En fonction des dépendances déclarées dans notre projet et en fonction de la valeur des propriétés présentes dans ce fichier, Spring Boot va adapter la création du contexte d'application.

Propriétés

- L'annotation @Value est utilisable sur un attribut ou un paramètre pour un type primitif ou une chaîne de caractères. Elle donne la valeur par défaut à injecter.

```
@Value("${database.uri}")
private String databaseUri;

@Value("${database.login}")
private String login;
@Value("${database.password}")
private String password;

 private Connection connection;
```

Values déclarées dans le fichier de propriété

database.uri = jdbc:mariadb://localhost:3306/db

database.login = root

database.password = r00t

Ajout de fichier de propriétés

- Une application Spring Boot supporte par défaut l'utilisation d'un fichier application.properties.
- Mais si vous ne souhaitez pas utiliser Spring Boot ou que vous voulez ajouter des fichiers de configuration supplémentaires, vous pouvez utiliser l'annotation @PropertySource pour désigner l'emplacement du fichier ou des fichiers de propriétés.

```
@SpringBootApplication
@PropertySource("classpath:config.properties")
public class MyApplication {
```

Ajout de fichier de propriétés

- L’emplacement du fichier est donné sous la forme d’une URI.
- schéma classpath désigne l’emplacement d’une fichier dans le chemin de classe.
- Mais il est également possible d’utiliser file pour désigner un chemin dans le système de fichiers ou même http et https pour télécharger un fichier de configuration depuis le Web.

```
@PropertySource({"classpath:config.properties", "file:config.properties"})
```

Classe Environment

- Si vous avez besoin de réaliser des traitements plus complexes à partir des propriétés, vous pouvez demander à injecter un *bean* de type Environment.
- Cette interface définie par le Spring Framework, vous permet d'accéder à la valeur des propriétés de votre application en appelant une de ses méthodes (telles que getProperty).

```
@Component
public class DemoProperty {
    @Autowired
    private Environment env;
    @PostConstruct
`   public void display() {
    System.out.println(env.getProperty("user.name"));
    }
}
```

Bean de propriétés

- Spring Boot propose une version avancée de la configuration d'une application.
- Il s'agit de représenter les données présentes dans un fichier de propriétés sous la forme d'un *bean* avec les annotations @Configuration et @ConfigurationProperties.
- Fichier de propriété

database.uri = jdbc:mariadb://localhost:3306/db

database.login = root

database.password = r00t

Bean de propriétés

```
@Configuration
@ConfigurationProperties(prefix = "database")
 public class DatabaseConfig {
    private String uri;
    private String login;
    private String password;
    // get and set obligatoire
}

@Component public class SimpleConnectionProvider{
    @Autowired private
    DatabaseConfig databaseConfig;
```

- Des questions?

Api REST avec Spring

API REST

- Services Web
- Echange de données en format Json
- Standard du web

En spring

```
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
@RestController
public class GreetingController {
    private static final String template = "Hello, %s!";
    @GetMapping("/greeting")
    public Greeting greeting(@RequestParam(value = "name", defaultValue =
"World") String name) {
        return new Greeting(counter.incrementAndGet(), String.format(template,
name));
    }
}
```

@Controller

- Cette annotation est utilisée pour créer une classe en tant que contrôleur Web, qui peut gérer les demandes des clients et renvoyer une réponse au client.
- Il s'agit d'une annotation au niveau de la classe, qui est placée au-dessus de votre classe de contrôleur.
- Semblable à @Service et @Repository, il s'agit également d'une annotation stéréotypée.

@RequestMapping

- La classe Controller contient plusieurs méthodes de gestionnaire pour gérer différentes requêtes HTTP, mais comment Spring mappe-t-il une requête particulière à une méthode de gestionnaire particulière ?
- A l'aide de l'annotation @RequestMapping.
  - C'est une annotation au niveau de la méthode qui est spécifiée sur une méthode de gestionnaire.
  - Il fournit le mappage entre le chemin de la demande et la méthode du gestionnaire.
  - Il prend également en charge certaines options avancées qui peuvent être utilisées pour spécifier des méthodes de gestion distinctes pour différents types de requêtes sur le même URI, comme vous pouvez spécifier une méthode pour gérer la requête GET et une autre pour gérer la requête POST sur le même URI.

@RequestMapping

- En termes simples, @RequestMapping marque les méthodes du gestionnaire de requêtes à l'intérieur des classes @Controller ; il peut être configuré en utilisant :
  - path ou ses alias, nom et valeur : l'URL à laquelle la méthode est mappée
  - method : méthodes HTTP compatibles
  - params : filtre les requêtes en fonction de la présence, de l'absence ou de la valeur des paramètres HTTP
  - headres : filtre les requêtes en fonction de la présence, de l'absence ou de la valeur des en-têtes HTTP
  - *consumes* : quels types de médias la méthode peut consommer dans le corps de la requête HTTP
  - *produces* : quels types de médias la méthode peut produire dans le corps de la réponse HTTP

  - De plus, @GetMapping, @PostMapping, @PutMapping, @DeleteMapping et @PatchMapping sont des variantes différentes de @RequestMapping avec la méthode HTTP déjà définie sur GET, POST, PUT, DELETE et PATCH respectivement.

@RequestParam

- Il s'agit d'une autre annotation Spring MVC utile qui est utilisée pour lier les paramètres HTTP aux arguments de méthode des méthodes de gestionnaire.
- Par exemple, si vous envoyez des paramètres de requête avec URLlikie pour la pagination ou simplement pour fournir des données clés, vous pouvez les obtenir en tant qu'arguments de méthode dans vos méthodes de gestionnaire.

@PathVariable

- Il s'agit d'une autre annotation utilisée pour récupérer des données à partir de l'URL.
- Contrairement à l'annotation @RequestParam qui est utilisée pour extraire les paramètres de requête, cette annotation permet au contrôleur de gérer une demande d'URL paramétrées comme les URL qui ont une entrée variable dans le cadre de leur chemin comme :
  - http://localhost:8080/books/900083838

```
@RequestMapping(value="/books/{ISBN}", method= RequestMethod.GET)
public String showBookDetails(@PathVariable("ISBN") String id, Model
model){
    model.addAttribute("ISBN", id); return "bookDetails";
}
```

@RequestBody

- Cette annotation peut convertir les données HTTP entrantes en objets Java transmis à la méthode de gestion du contrôleur.
- Tout comme @ResponseBody indique au Spring MVC d'utiliser un convertisseur de message lors de l'envoi d'une réponse au client, les annotations @RequestBody indiquent au Spring de trouver un convertisseur de message approprié pour convertir une représentation de ressource provenant d'un client en un objet.

```
@RequestMapping(method=RequestMethod.POST, consumers=
"application/json")
public @ResponseBody Course saveCourse(@RequestBody Course
aCourse){
    return courseRepository.save(aCourse);
}
```

@ResponseBody

- L'annotation @ResponseBody est l'une des annotations les plus utiles pour développer un service Web RESTful à l'aide de Spring MVC.
- Cette annotation est utilisée pour transformer un objet Java renvoyé par un contrôleur en une représentation de ressource demandée par un client REST.

```
@RequestMapping(method=RequestMethod.POST,consumers=
"application/json")
public @ResponseBody Course saveCourse(@RequestBody Course
aCourse){
    return courseRepository.save(aCourse);
}
```

@ResponseStatus

- Cette annotation peut être utilisée pour remplacer le code de réponse HTTP pour une réponse.
- Vous pouvez utiliser cette annotation pour la gestion des erreurs lors du développement d'une application Web ou d'un service Web RESTful à l'aide de Spring.

```
@Exceptionhandler(BookNotFoundException.class)
@ResponseStatus(HttpStatus.NOT_FOUND)
public Error bookNotFound(BookNotFoundException bnfe){ l
    ong ISBN = bnfe.getISBN();
    return new Error(4, "Book [" + ISBN + "] not found");
}
```

ResponseEntity

- La classe ResponseEntity est utilisée lorsque nous spécifions par programme tous les aspects d'une réponse HTTP.
- Cela inclut les en-têtes, le corps et, bien sûr, le code d'état.
- Cette méthode est la manière la plus détaillée de renvoyer une réponse HTTP dans Spring Boot, mais aussi la plus personnalisable.
- Beaucoup préfèrent utiliser l'annotation @ResponseBody couplée à @ResponseStatus car elles sont plus simples.

```
@GetMapping("/response_entity")
public ResponseEntity<String> withResponseEntity() {
    return
ResponseEntity.status(HttpStatus.CREATED).body("HTTP Status
will be CREATED (CODE 201)\n");
}
```

Controller exemple

```java
@RestController
@RequestMapping("/api/v1")
public class UserController {

    @Autowired
    private UserRepository userRepository;

    @GetMapping("/users")
    public List<User> getAllUsers() {
        return userRepository.findAll();
    }

    @GetMapping("/users/{id}")
    public ResponseEntity<User> getUserById(
    @PathVariable(value = "id") Long userId) throws ResourceNotFoundException {
        User user = userRepository.findById(userId)
        .orElseThrow(() -> new ResourceNotFoundException("User not found on :: "+ userId));
        return ResponseEntity.ok().body(user);
    }

    @PostMapping("/users")
    public User createUser(@Valid @RequestBody User user) {
        return userRepository.save(user);
```

Service et Repository

- Le contrôleur doit faire appel au Service pour accéder aux données.
- Le Service lui fait appel au Repository.
- La présence du Repository nous permet également d'avoir une persistance de la donnée, par exemple via une base de données SQL.

```
public interface BookService {
  Book getBookById(Integer bookId);
  void addBook(Book book);
  List<Book> getAllBooks();
}
public interface BookRepository {
  Book getBookById(Integer bookId);
  void addBook(Book book);
  List<Book> getAllBooks();
}
```

Injection dans le controlleur

```
private BookService bookService;

@Autowired
public BookController(final BookService bookService){
  this.bookService = bookService;
}
```

Intercepteur

- Permettent d'exécuter du code sur chaque requête de façon à initialiser des paramètres globaux, par exemple des en-tête HTTP comme la langue (Accept-Language).
- En utilisant les intercepteurs, on peut éviter la répétition de code par exemple lors de la gestion d'authentification avec un jeton OAuth2.
- Au lieu de l'avoir dans chaque contrôleur, il suffit d'avoir un intercepteur qui s'occupe de traiter le jeton pour chaque requête et de placer l'information dans un objet qui pourra être injecté aux classes qui en ont besoin.

Exemple

- Paramètre user

```
@Component
public class ApplicationContext {
  private String user;
  public String getUser() {return user; }
  public void setUser(String user) {this.user = user; }
}
```

- L'annotation @Component nous permettra d'injecter un objet ApplicationContext dans les classes qui en ont besoin.

Exemple

- On peut ensuite créer la classe HeadersInterceptor.
- En surchargeant la méthode preHandle de sa classe parent HandlerInterceptorAdapter, on peut accéder aux information de la requête HTTP et modifier l'objet ApplicationContexte qui a été injecté :

```
@Override
public boolean preHandle(final HttpServletRequest request, final
HttpServletResponse response, final Object handler) {
  final Enumeration<String> headers =
request.getHeaders("user");
  final List<String> users = Collections.list(headers);

  if(!users.isEmpty()){
    String userName = users.get(0);
    this.applicationContext.setUser(userName);
  }

  return true;
}
```

Exemple

- Il faut ensuite enregistrer l'intercepteur dans la classe InterceptorConfig qui hérite de WebMvcConvigurer :

```
@Configuration
public class InterceptorsConfig implements WebMvcConfigurer {

  @Autowired
  private HeadersInterceptor headersInterceptor;

  @Override
  public void addInterceptors(InterceptorRegistry registry) {
    registry.addInterceptor(this.headersInterceptor);
  }
}
```

- L'annotation @Configuration indique à Spring que cette classe contient de la configuration globale, et le code contenu dans la classe sera exécuté au démarrage de l'application.

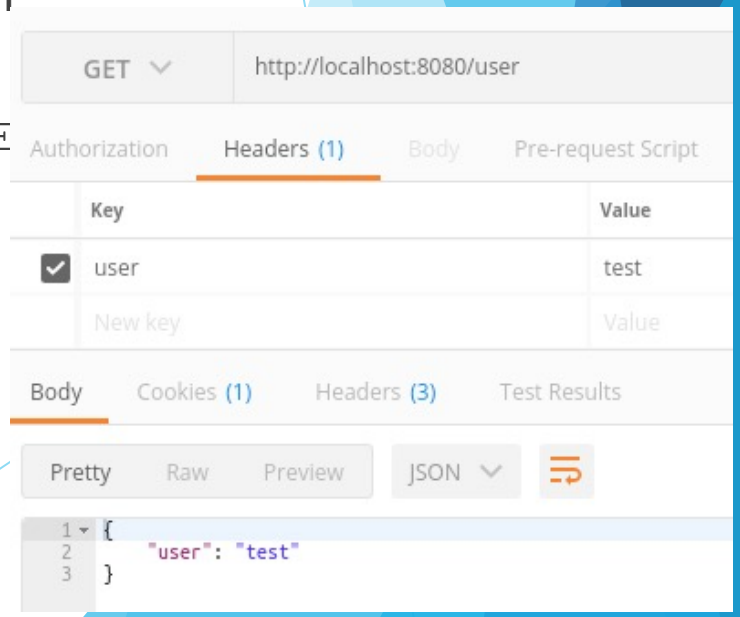
Exemple

- Il serait également possible d'indiquer que l'intercepteur ne concerne que certaines routes et doit en exclure d'autre:

```
registry.addInterceptor(this.headersInterceptor)
.addPathPatterns("/users/**")
.excludePathPatterns("/users/public/**");
```

- Afin de tester l'intercepteur de façon concrète, nous pouvons créer un contrôleur simple (UserController) qui retourne simplement l'information contenue dans l'objet ApplicationContext :

```
@RequestMapping(value = "/user", method = RequestMethod.GE
@ResponseBody
ApplicationContext getContext() {
  return this.applicationContext;
}
```



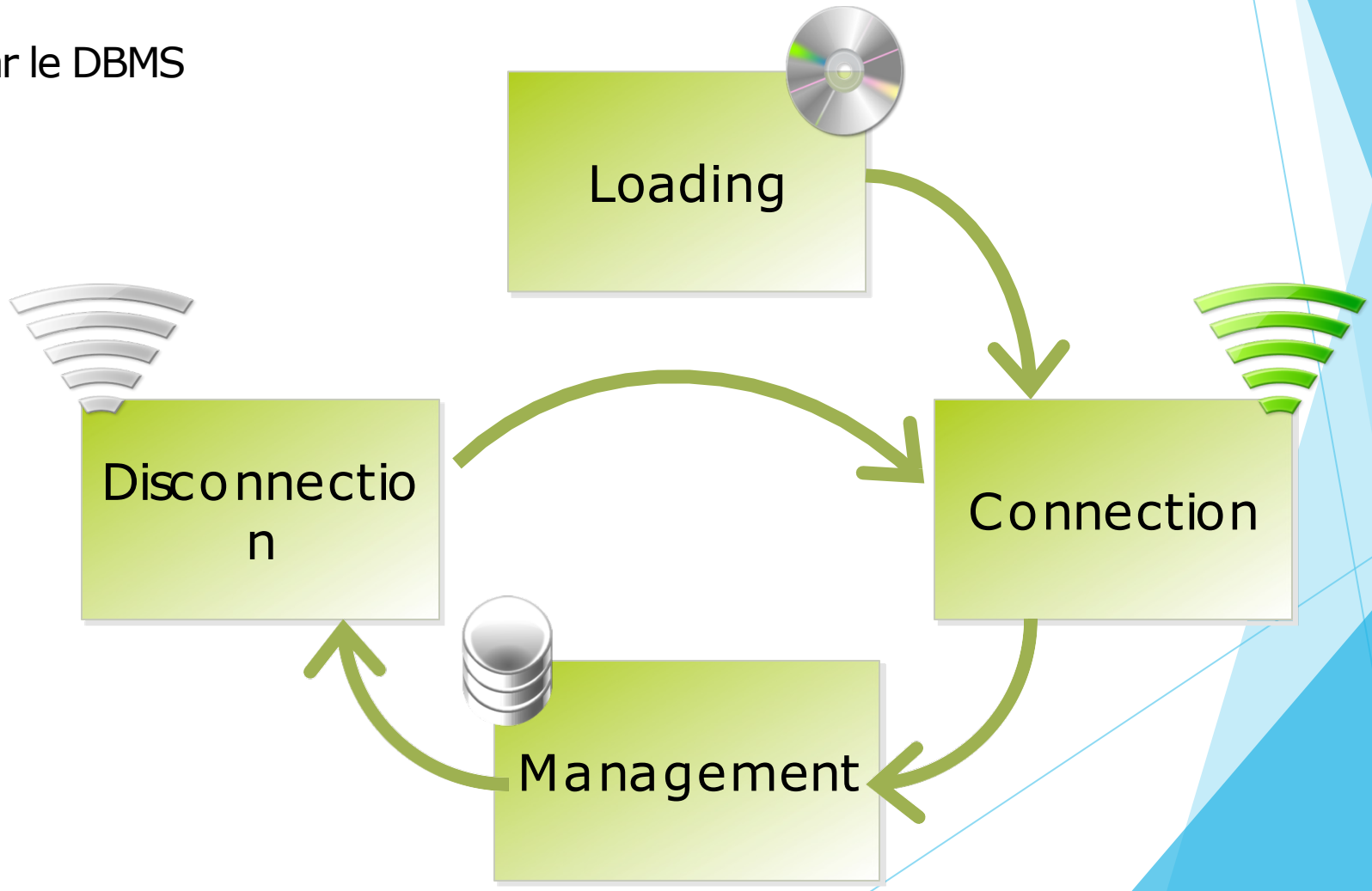
- Des questions?

Persistance avec JDBC

Losrque l'application a besoin de performances élevées

JDBC

- Gestion de la connexion au DBMS
- Fourni par le DBMS

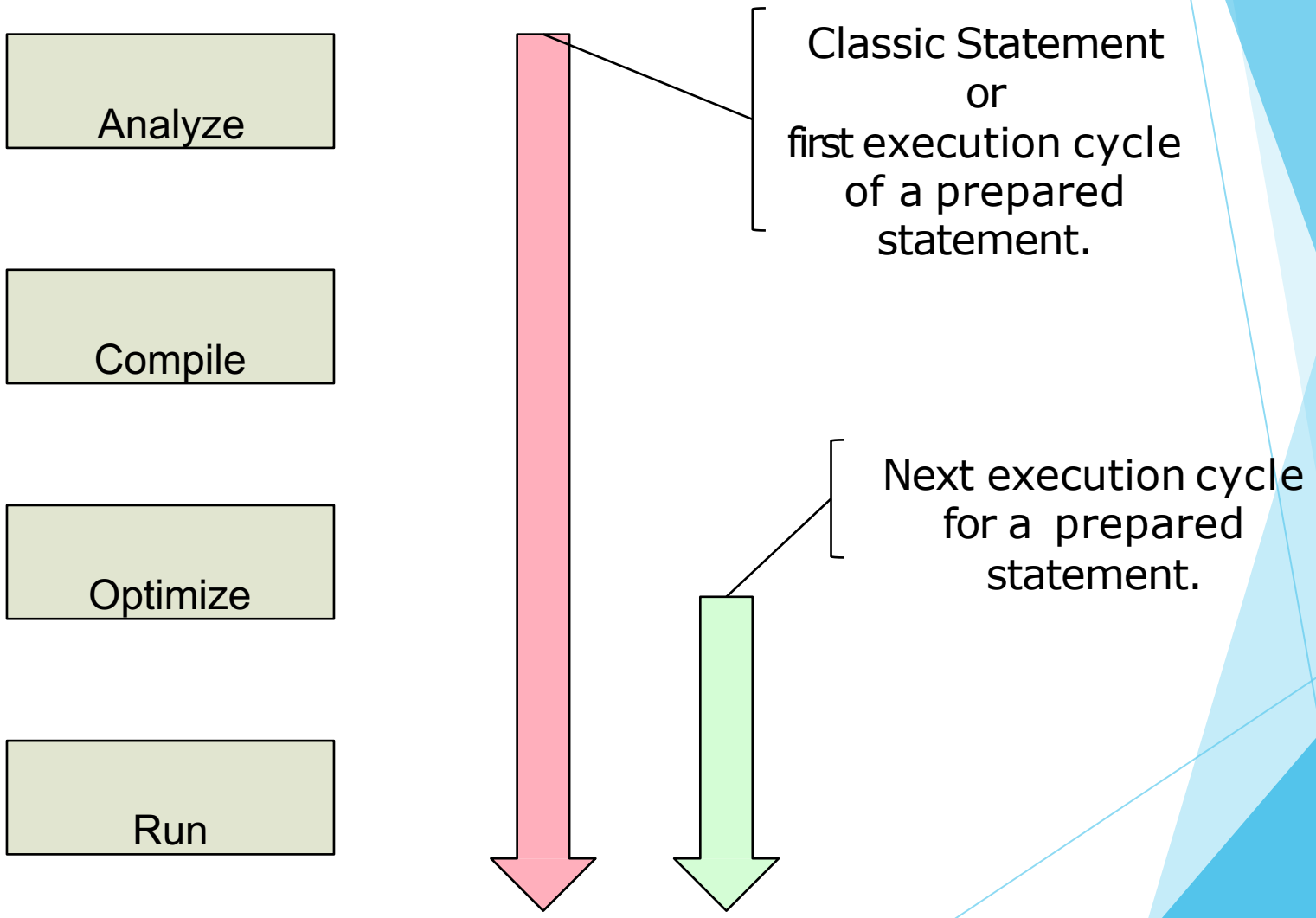


Statement

- One interface.
- With **java.sql.Connection**:
  - **createStatement()**
- Make the request
  - **ResultSet** executeQuery(String query)
  - **int** executeUpdate(String query)

```
Statement stmt =myConnection.createStatement();
ResutSet rs =stmt.executeQuery("SELECT *FROM dummy");
```

PreparedStatement



PreparedStatement

- **With java.sql.Connection :**
  - **prepareStatement(String query)**

- Methods :
  - **setInt(int index, int value)**
  - …

```
PreparedStatement pstmt =
myConnection.prepareStatement("SELECT *FROM dummy
WHERE lastname =? AND firstname =?");
pstmt.setString(1, "GEORGES");
pstmt.setString(2, "Ron");
```

ResultSet

- Request result.
- Iteration :
  - **boolean previous()**
  - **boolean next()**.
- Get a data colunm:
  - **getXXX(int columnIndex)**
  - **getXXX(String columnName)**

ResultSet

- Example :

```
// ...
ResultSet rs =stmt.executeQuery("SELECT *FROM dummy");

while(rs.next()) {
        int id =rs.getInt(1);
        String name =rs.getString("name");
        // ...
}
// ...
```

Transactions

- **void setAutoCommit (boolean) :**
  - Actived par default.
- **void commit ( )**
- **void rollback ( )**

Spring JDBC

- Graddle

```
plugins {
    id 'org.springframework.boot' version '2.5.2’
    id 'io.spring.dependency-management' version '1.0.11.RELEASE' id 'java' }
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-jdbc’
    runtimeOnly 'com.h2database:h2’
    testImplementation 'org.springframework.boot:spring-boot-starter-test’ }
```

- Propriété du driver

```
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/mydatabase
spring.datasource.username=root
spring.datasource.password=12345
```

Accès Base

- Spring fournit une classe de modèle appelée JdbcTemplate qui facilite l'utilisation des bases de données relationnelles SQL et JDBC.
- La plupart du code JDBC est embourbé dans l'acquisition de ressources, la gestion des connexions, la gestion des exceptions et la vérification générale des erreurs qui n'ont aucun rapport avec ce que le code est censé réaliser.
- Le JdbcTemplate s'occupe de tout cela pour vous.
- Tout ce que vous avez à faire est de vous concentrer sur la tâche à accomplir.

Entité = JavaBeans

```
public class Employee {
    private long id; private String firstName, lastName;
    public Employee(long id, String firstName, String lastName) {
        this.id = id; this.firstName = firstName; this.lastName =
    lastName;
    }
    @Override public String toString() {
        return String.format( " Employee[id=%d, firstName='%s',
            lastName='%s']", id, firstName, lastName);
    }
    // getters & setters
}
```

Exemple

```
@SpringBootApplication
public class RelationalDataAccessApplication implements CommandLineRunner {


    public static void main(String args[]) {
          SpringApplication.run(RelationalDataAccessApplication.class, args); }
    @Autowired
    JdbcTemplate jdbcTemplate;
    @Override
    public void run(String... strings) throws Exception {
        jdbcTemplate.execute("DROP TABLE customers IF EXISTS");
        jdbcTemplate.execute("CREATE TABLE customers(" + "id SERIAL, first_name VARCHAR(255), last_name
        VARCHAR(255))");
        // Split up the array of whole names into an array of first/last names
        List<Object[]> splitUpNames = Arrays.asList("John Woo", "Jeff Dean", "Josh Bloch", "Josh Long")
          .stream() .map(name -> name.split(" ")) .collect(Collectors.toList());

        // Uses JdbcTemplate's batchUpdate operation to bulk load data
        jdbcTemplate.batchUpdate("INSERT INTO customers(first_name, last_name) VALUES (?,?)",
        splitUpNames);
        jdbcTemplate.query( "SELECT id, first_name, last_name FROM customers WHERE first_name = ?", new
        Object[] { "Josh" }, (rs, rowNum) -> new Customer(rs.getLong("id"),
        rs.getString("first_name"), rs.getString("last_name")) ).forEach(customer ->
        log.info(customer.toString()));
    }
}
```

Configuration

```
@Configuration @ComponentScan("com.example.jdbc")
    public class SpringJdbcConfig {
    @Bean public DataSource mysqlDataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("com.mysql.jdbc.Driver");
        dataSource.setUrl("jdbc:mysql://localhost:3306/springjdbc");
        dataSource.setUsername("guest_user");
        dataSource.setPassword("guest_password");
        return dataSource;
    }
}
```

Configuration xml

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
destroy-method="close">
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql://localhost:3306/springjdbc"/>
    <property name="username" value="guest_user"/>
    <property name="password" value="guest_password"/>
</bean>
```

Requêtes

```
int result = jdbcTemplate.queryForObject( "SELECT COUNT(*)
FROM EMPLOYEE", Integer.class);

public int addEmplyee(int id) {

   return jdbcTemplate.update( "INSERT INTO EMPLOYEE VALUES
(?, ?, ?, ?)", id, "Bill", "Gates", "USA");

}
```

Méthodes : execute(), query(), …

Named paramètres

```
SqlParameterSource namedParameters = new
MapSqlParameterSource().addValue("id", 1);

namedParameterJdbcTemplate.queryForObject( "SELECT FIRST_NAME
FROM EMPLOYEE WHERE ID = :id", namedParameters, String.class);


Employee employee = new Employee();
employee.setFirstName("James");

String SELECT_BY_ID = "SELECT COUNT(*) FROM EMPLOYEE WHERE
FIRST_NAME = :firstName";

SqlParameterSource namedParameters = new
BeanPropertySqlParameterSource(employee);

namedParameterJdbcTemplate.queryForObject( SELECT_BY_ID,
namedParameters, Integer.class);
```

Mapping

- Interface RowMapper

```
public class EmployeeRowMapper implements RowMapper<Employee> {
    @Override public Employee mapRow(ResultSet rs, int rowNum) throws SQLException {
    Employee employee = new Employee();
    employee.setId(rs.getInt("ID"));
    employee.setFirstName(rs.getString("FIRST_NAME"));
    employee.setLastName(rs.getString("LAST_NAME"));
    return employee;
    }
}
```

```
String query = "SELECT * FROM EMPLOYEE WHERE ID = ?";
Employee employee = jdbcTemplate.queryForObject(
    query, new Object[] { id }, new EmployeeRowMapper());
```

- Des questions?

Persistance avec JPA

Lorsque les données sont simples et bien organisées.

Configuration

- Graddle

```
plugins {
    id 'org.springframework.boot' version '2.5.2’
    id 'io.spring.dependency-management' version '1.0.11.RELEASE' id 'java’ }
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-data-jpa’
    runtimeOnly 'com.h2database:h2’
    testImplementation 'org.springframework.boot:spring-boot-starter-test' }
```

Annotation

```
public class Contact implements Serializable {
    // my properties
    private int id;
    private String name;
    private String firstname;

    // ... setters and getters ...
}
```

Entity

```
@Entity
@Table(name="CONTACTS"
)
public class Contact  implements Serializable {

}
```

@Id

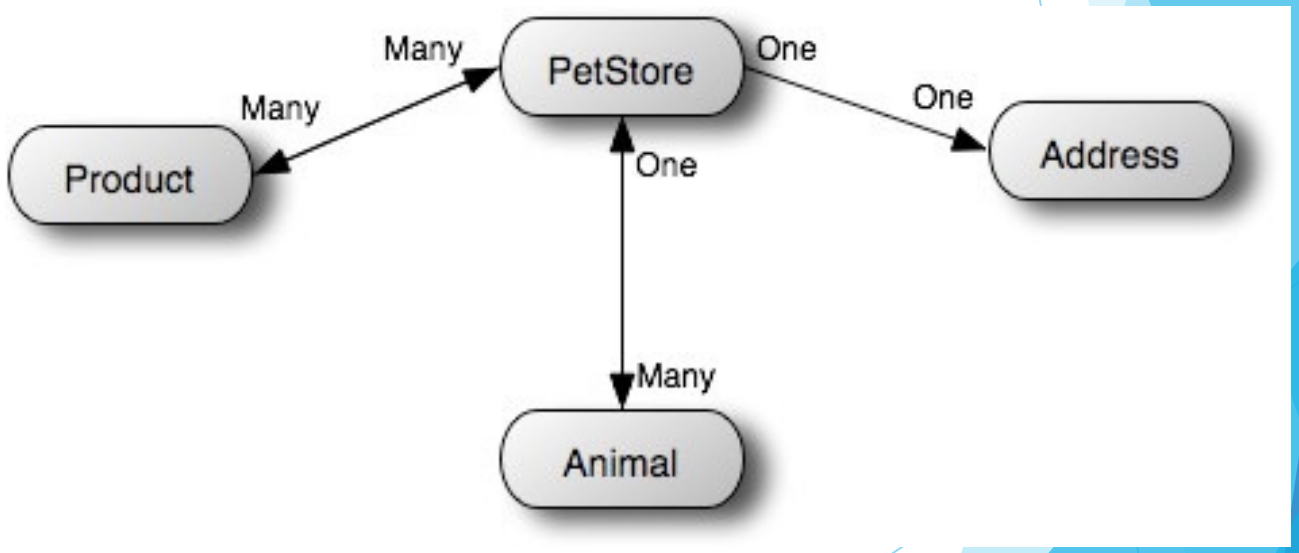
- **Auto increment**
  - **@GeneratedValue**(strategy=GenerationType.XXX)
- Avec comme type:
  - IDENTITY
  - SEQUENCE
  - TABLE
  - AUTO

Annotations

Annotation	Description
@Basic	basique
@Transient	No save
@Lob	Blob
@Temporal	Date and time
@Enumerated	Enumeration

Join

- One-To-One
- One-To-Many
- Many-To-One
- Many-To-Many



One-To-One

3 politiques:

- @JoinColumn
  - Clé étrangère
- @PrimaryKeyJoinColumn
  - Même clé primaire
- @JoinTable
  - Table de jointure

One-To-One

```
public class PetStore {
   ...
   @OneToOne
   @JoinColumn(name="address_fk"
   )  private      Address
            address;
   ...
}
```

One-To-One and Many-To-One

- Table de jointure ou clé étrangère
  - @JoinTable
  - @JoinColumn

One-To-One and Many-To-One

Store Entity

```
@OneToMany(mappedBy="petStore")
private Collection<Animal> animals;
```

Animal Entity

```
@ManyToOne
@JoinColumn(name="store_fk")
private PetStore petStore;
```

Many-ToMany

- @JoinTable uniquement

Many-To-Many

Store Entity	`@ManyToMany`
	`@JoinTable(name="STORE_PRODUCT")`
	`private Collection<Product> products;`

Product Entity	`@ManyToMany(mappedBy="products")` `private Collection<PetStore> stores;`

Cascading

- SQL cascade
- Quatre types:
  - `PERSIST|MERGE|REMOVE|REFRESH`
  - `CascadeType.ALL`

Cascading

PetStore Entity	

```
@OneToOne(cascade=CascadeType.PERSIST
)  @JoinColumn(name="address_fk")
private Address address;
```

Lazy loading

- Par défault, les relations 1 à n ou n à n, les objets des compositions ne sont pas chargés automatiquement : LAZY
- 2 Types : `LAZY|EAGER`

Lazy loading

PetStore Entity	`@OneToMany(mappedBy="petStore"`
	`, fetch=FetchType.EAGER)`
	`private Collection<Animal> animals;`

Inheritance

- Les bases de données relationnelles ne disposent pas d'un moyen simple de mapper les hiérarchies de classes sur les tables de base de données.
- Pour résoudre ce problème, la spécification JPA propose plusieurs stratégies :
  - MappedSuperclass - les classes parentes, ne peuvent pas être des entités
  - Single Table - les entités de différentes classes avec un ancêtre commun sont placées dans une seule table
  - Joined Table - chaque classe a sa table et l'interrogation d'une entité de sous-classe nécessite de joindre les tables
  - Table-Per-Class - toutes les propriétés d'une classe sont dans sa table, donc aucune jointure n'est requise
- Chaque stratégie se traduit par une structure de base de données différente.
- L'héritage d'entité signifie que nous pouvons utiliser des requêtes polymorphes pour récupérer toutes les entités de la sous-classe lors de la recherche d'une super-classe.

MappedSuperclass

```java
@MappedSuperclass
public class Person {

    @Id
    private long personId;
    private String name;

    // constructor, getters, setters
}
```

```java
@Entity
public class MyEmployee extends Person {
    private String company;
    // constructor, getters, setters
}
```

Single table

```java
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
public class MyProduct {
    @Id
    private long productId;
    private String name;

    // constructor, getters, setters
}
```

```java
@Entity
public class Book extends MyProduct {
    private String author;
}
```

```java
@Entity
public class Pen extends MyProduct {
    private String color;
}
```

Discriminator colunm

```
@Entity(name="products")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="product_type",
  discriminatorType = DiscriminatorType.INTEGER)
public class MyProduct {
    // ...
}
```

```
@Entity
@DiscriminatorValue("1")
public class Book extends MyProduct {
    // ...
}

@Entity
@DiscriminatorValue("2")
public class Pen extends MyProduct {
    // ...
}
```

Joined table

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class Animal {
    @Id
    private long animalId;
    private String species;

    // constructor, getters, setters
}
```

```
@Entity
public class Pet extends Animal {
    private String name;

    // constructor, getters, setters
}
```

```
@Entity
@PrimaryKeyJoinColumn(name = "petId")
public class Pet extends Animal {
    // ...
}
```

Table per class

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Vehicle {
    @Id
    private long vehicleId;

    private String manufacturer;

    // standard constructor, getters, setters
}
```

Repository

```java
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import com.companyname.springbootcrudrest.model.User;

@Repository
public interface UserRepository extends JpaRepository<User, Long>{

}
```

- Notez que nous avons annoté l'interface avec l'annotation @Repository. Cela indique à Spring d'amorcer le référentiel lors d'une analyse de composant.
- L'interface UserRepository étend JpaRepository qui fournit les méthodes ci-dessous pour gérer les opérations de base de données :

```java
List<T> findAll(Sort sort);
List<T> findAllById(Iterable<ID> ids);
<S extends T> List<S> saveAll(Iterable<S> entities);
void flush();
<S extends T> S saveAndFlush(S entity);
void deleteInBatch(Iterable<T> entities);
void deleteAllInBatch();
T getOne(ID id);
@Override
<S extends T> List<S> findAll(Example<S> example);
<S extends T> List<S> findAll(Example<S> example, Sort sort);
```

Méthodes du repo

Keyword	Sample	JPQL snippet
Distinct	findDistinctByLastnameAndFirstname	select distinct … where x.lastname = ?1 and x.firstname = ?2
And	findByLastnameAndFirstname	… where x.lastname = ?1 and x.firstname = ?2
Or	findByLastnameOrFirstname	… where x.lastname = ?1 or x.firstname = ?2
Is, Equals	findByFirstname,findByFirstnameIs,findByFirstnameEquals	… where x.firstname = ?1
Between	findByStartDateBetween	… where x.startDate between ?1 and ?2
LessThan	findByAgeLessThan	… where x.age < ?1
LessThanEqual	findByAgeLessThanEqual	… where x.age <= ?1
GreaterThan	findByAgeGreaterThan	… where x.age > ?1
GreaterThanEqual	findByAgeGreaterThanEqual	… where x.age >= ?1

Méthodes du repo

Keyword	Sample	JPQL snippet
After	findByStartDateAfter	… where x.startDate > ?1
Before	findByStartDateBefore	… where x.startDate < ?1
IsNull, Null	findByAge(Is)Null	… where x.age is null
IsNotNull, NotNull	findByAge(Is)NotNull	… where x.age not null
Like	findByFirstnameLike	… where x.firstname like ?1
NotLike	findByFirstnameNotLike	… where x.firstname not like ?1
StartingWith	findByFirstnameStartingWith	… where x.firstname like ?1 (parameter bound with appended %)
EndingWith	findByFirstnameEndingWith	… where x.firstname like ?1 (parameter bound with prepended %)
Containing	findByFirstnameContaining	… where x.firstname like ?1 (parameter bound wrapped in %)

Méthodes du repo

Keyword	Sample	JPQL snippet
OrderBy	findByAgeOrderByLastnameDesc	… where x.age = ?1 order by x.lastname desc
Not	findByLastnameNot	… where x.lastname <> ?1
In	findByAgeIn(Collection<Age> ages)	… where x.age in ?1
NotIn	findByAgeNotIn(Collection<Age> ages)	… where x.age not in ?1
True	findByActiveTrue()	… where x.active = true
False	findByActiveFalse()	… where x.active = false
IgnoreCase	findByFirstnameIgnoreCase	… where UPPER(x.firstname) = UPPER(?1)

Custom repo

- Pour ajouter des requêtes non supportées par le repo classique.

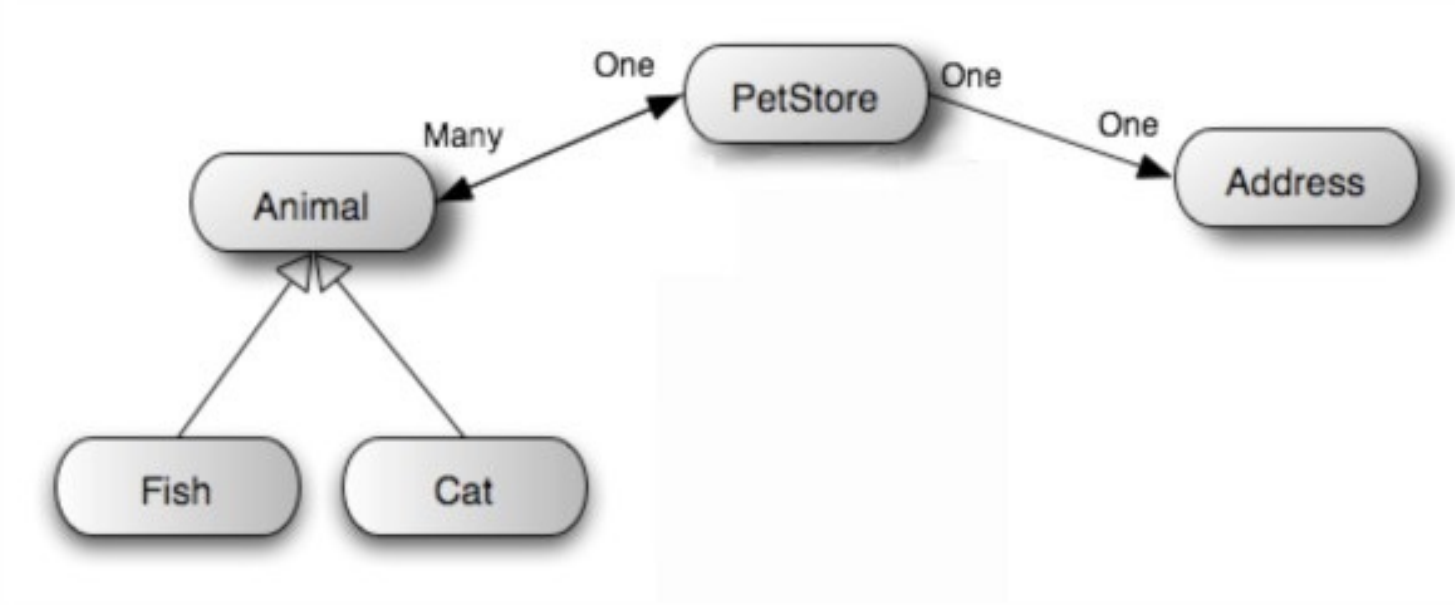
```
@Repository
@Transactional
public class BookRepository {

        @PersistenceContext
        private EntityManager em;
        public Book get(int id) {
                Book res = null;
                try {
                        res = em.find(Book.class, id);
                } catch (Exception e) {
                        e.printStackTrace();
                }
                return res;
        }
```

Entity Manager

- Méthodes :
  - void persist(Object entity)
  - <T>Tmerge(T entity)
  - void remove(Object entity)
  - <T>Tfind(Class<T> entityClass, Object primaryKey)

JPQL

Request on class and not tables



SELECT statement

```
Query query = em.createQuery("SELECT c FROM Cat AS c");
List<Cat>list = query.getResultList();
```

WHERE clause

```
Query  query = em.createQuery("SELECT cat FROM Cat AS cat
   WHERE cat.animalId = 5");

Cat myCat = (Cat)query.getSingleResult();
```

- IS NULL
- ...

DELETE and UPDATE statements

```
Query  query = em.createQuery("DELETE FROM Cat AS cat WHERE
   cat.earLength = 2");

int nbrDeleted = query.executeUpdate();
```

```
Query  query = em.createQuery("UPDATE Cat AS cat SET
   cat.earLength = 3 WHERE cat.earLength = 4");

int nbrUpdated = query.executeUpdate();
```

Parameters

```
Query query = em.createQuery("SELECT cat FROM Cat AS cat
   WHERE cat.animalId = ?1");
query.setParameter(1, 5);
Cat myCat = (Cat)query.getSingleResult();
```

```
Query query = em.createQuery("SELECT cat FROM Cat AS cat
   WHERE cat.animalId = :id");
query.setParameter("id", 5);
Cat myCat = (Cat)query.getSingleResult();
```

Aggregation

- Aggregation functions can be used with the SELECT clause
  - MIN
  - AVG
  - …

```
Query query = em.createQuery("SELECT MAX(cat.earLength) FROM
   Cat AS cat");

Number maxEarLength = (Number) query.getSingleResult();
```

- Des questions?

Spring actuators

Actuator

- **Spring Boot Actuator** ét un sous projet (sub-project) du projet **Spring Boot**.
- Il est créé pour collecter, superviser les informations de l'application.
- Graddle

```
plugins {
    id 'org.springframework.boot' version '2.5.2’
     id 'io.spring.dependency-management' version '1.0.11.RELEASE' id 'java' }
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-actuator’
    implementation 'org.springframework.boot:spring-boot-starter-web’
    testImplementation 'org.springframework.boot:spring-boot-starter-test' }
```

Actuator

- En substance, Actuator apporte à notre application des fonctionnalités prêtes pour la production.
- La surveillance de notre application, la collecte de métriques, la compréhension du trafic ou l'état de notre base de données deviennent triviales avec cette dépendance.
- Le principal avantage de cette bibliothèque est que nous pouvons obtenir des outils de production sans avoir à implémenter ces fonctionnalités nous-mêmes.
- Actuator est principalement utilisé pour exposer des informations opérationnelles sur l'application en cours d'exécution - santé, métriques, informations, dump, env, etc. Il utilise des points de terminaison HTTP ou des beans JMX pour nous permettre d'interagir avec elle.
- Une fois que cette dépendance est sur le chemin de classe, plusieurs points de terminaison sont disponibles pour nous hors de la boîte.
- Comme avec la plupart des modules Spring, nous pouvons facilement le configurer ou l'étendre de plusieurs manières.

Remaque avec security

- Actuator partage la configuration de sécurité avec les règles de sécurité habituelles de l'application, de sorte que le modèle de sécurité est considérablement simplifié.
- Par conséquent, pour modifier les règles de sécurité de l'actionneur, nous pourrions simplement ajouter une entrée pour /actuator/** :

```
@Bean public SecurityWebFilterChain securityWebFilterChain(
ServerHttpSecurity http) {
    return http.authorizeExchange()
        .pathMatchers("/actuator/**").permitAll()
        .anyExchange().authenticated() .and().build();
}
```

Endpoints

- Dans une application Spring Boot, nous exposons un point de terminaison d'API REST en utilisant l'annotation @RequestMapping dans la classe du contrôleur.
- Actuator permet de récupérer des informations sur ces endpoints.

- Application.properties

management.endpoints.web.exposure.include=*

management.endpoint.xxx.cache.time-to-live=10s //*temps de réponse en cache*

management.endpoints.web.discovery.enabled=false

// CORS

management.endpoints.web.cors.allowed-origins=https://example.com
management.endpoints.web.cors.allowed-methods=GET,POST

- Informations : http://localhost:9001/actuator.
- Deux endpoints sont publics par défaut : /health et /info.
  - { "status": "UP"} ou { "status": »DOWN"}

Endpoints prédéfini

- */beans*
  - envoie tous les beans disponibles dans notre BeanFactory
- */conditions*
  - anciennement connu sous le nom de /autoconfig, crée un rapport des conditions autour de la configuration automatique.
- */configprops*
  - nous permet de récupérer tous les beans @ConfigurationProperties.
- */env*
  - renvoie les propriétés actuelles de l'environnement. De plus, nous pouvons récupérer des propriétés individuelles.
- */health*
  - résume l'état de santé de notre application
- */heapdump*
  - construit et renvoie un vidage de tas à partir de la JVM utilisée par notre application.

Endpoints prédéfini

- */info*
  - renvoie des informations générales. Il peut s'agir de données personnalisées, d'informations de build ou de détails sur le dernier commit.
- */logfile*
  - renvoie les journaux d'application ordinaires.
- */loggers*
  - nous permet d'interroger et de modifier le niveau de journalisation de notre application.
- */metrics*
  - détaille les métriques de notre application. Cela peut inclure des métriques génériques ainsi que des métriques personnalisées.
- */scheduledtasks*
  - fournit des détails sur chaque tâche planifiée dans notre application.
- */sessions*
  - répertorie les sessions HTTP étant donné que nous utilisons Spring Session.
- */shutdown*
  - effectue un arrêt progressif de l'application.
- */threaddump*
  - vide les informations de thread de la JVM sous-jacente.

Custom indicateur

- Exemple : health
- Hériter le controlleur de HealthIndicator et redéfinir la éthode health()

```
@RestController
public class ProductController implements HealthIndicator {
    @Override
    public Health health() {
        …
        if(problem){
            return Health.down().build();
        }
        return Health.up().build();
    }
```

info

- Application.propreties

info.app.version=1.0-Beta

- Résultat


```
{
  "app": {
      "version": "1.0-Beta"
  }
}
```

Custom endpoint

- Annotations @Endpoint, @WebEndpoint ou @EndpointWebExtension
- Path par défaut : /actuator/id_actuator

Operation	HTTP method
@ReadOperation	GET
@WriteOperation	POST
@DeleteOperation	DELETE

Exemple

- management.endpoints.web.exposure.include=helloworld
- http://localhost:8080/actuator/helloworld
- http://localhost:8080/actuator/helloworld/name=toto
- http://localhost:8080/actuator/helloworld/toto

```
@Endpoint(id = "helloworld")
public class HelloWorldEndpoint {

	@ReadOperation
		public String helloWorld() {
		return "Hello World";
	}

	@ReadOperation
	public String helloName(String name) {
		return "Hello " + name;
	}

	@ReadOperation
	public String helloNameSelector(@Selector String name) {
		return "Hello " + name;
	}
}

@Configuration
public class CustomActuatorConfiguration {

	@Bean
	public HelloWorldEndpoint helloWorldEndpoint() {
		return new HelloWorldEndpoint();
	}

}
```

Exemple post & delete

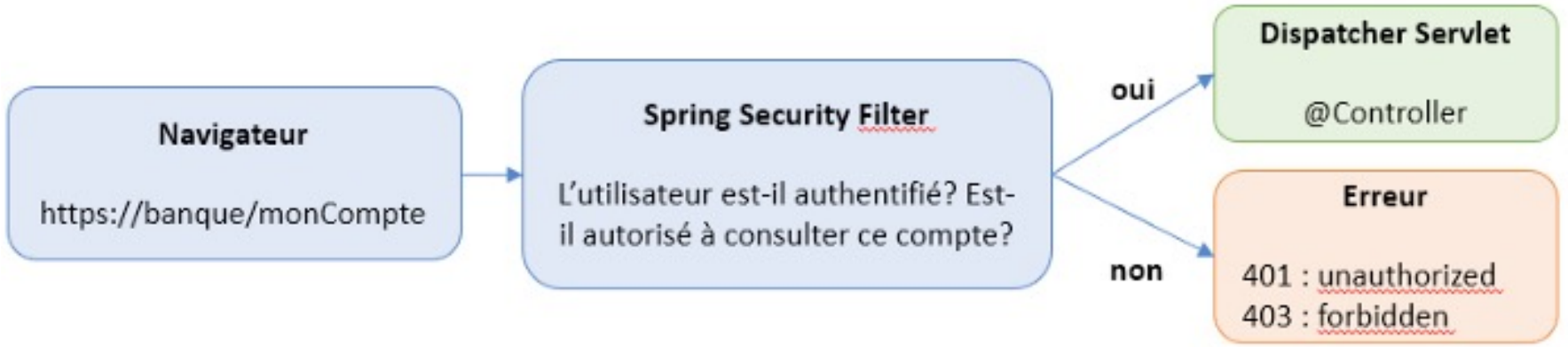
```
@WriteOperation
public String helloNameBody(String name) {
    return "Hello " + name;
}

@DeleteOperation
public String goodbyeNameParam(String name) {
    return "Goodbye " + name;
}

@DeleteOperation
public String goodbyeNameSelector(@Selector String name) {
    return "Goodbye " + name;
}
```

- Des questions?

Spring security pour les API

Spring security

- Sécurité applicative orchestrée par la *Dispatcher Servlet* de Spring, qui permet d'adresser les requêtes aux différents *controllers* de l'application.
- Spring Security ne fait qu'ajouter des traitements à cette orchestration, par le biais de *Servlet Filters*. L'ensemble des *Servlet Filters* constitue la **Filter Chain** de Spring Security.
- L'action de la *filter chain* est centrée autour de 2 concepts fondamentaux :
  - l'**authentification** : celui qui utilise l'application doit être identifié par un couple **username/password**.
  - les **autorisations** : tous les utilisateurs n'ont pas nécessairement accès aux mêmes fonctionnalités. Par exemple, un utilisateur non administrateur ne doit pas pouvoir modifier de compte autre que le sien.



graddle

```
plugins {
    id 'org.springframework.boot' version '2.5.2’
    id 'io.spring.dependency-management' version '1.0.11.RELEASE’
    id 'java’ }

dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-web’
    implementation 'org.springframework.boot:spring-boot-starter-security’
    implementation 'org.springframework.security:spring-security-test’
    testImplementation('org.springframework.boot:spring-boot-starter-test’)
}
```

Filter Chain par défaut

- **WebAsyncManagerIntegrationFilter**
  - fournit l'intégration entre le *SecurityContext* et le *WebAsyncManager* de Spring permettant d'alimenter le *SecurityContext* lors d'une requête.
- **SecurityContextPersistenceFilter**
  - place les informations obtenues du *SecurityContextRepository* dans le *SecurityContextHolder*, utilisé par la suite lors du process d'authentification qui nécessite un *SecurityContext* valide.
- **HeaderWriterFilter**
  - permet d'ajouter des headers à la réponse en cours de confection. Utile pour ajouter certains headers qui renforcent la sécurité au niveau du navigateur, comme X-Frame-Options, X-XSS-Protection et X-Content-Type-Options (cf. le paragraphe sur la protection générale).
- **CsrfFilter**
  - applique une protection contre les attaques de type *Cross-Site Request Forgery* en utilisant un token, usuellement stocké dans la *HttpSession*. Il est souvent demandé aux développeurs d'invoquer ce filtre avant toute requête susceptible de modifier l'état de l'application (usuellement les requête de type POST, PUT, DELETE et parfois OPTIONS).
- **LogoutFilter**
  - intervient lorsqu'une requête de déconnexion est reçue, et délègue les différents processus de déconnexion à divers *lougoutHandlers* (nettoyage du contexte de sécurité, invalidation de la session, redirection…).

Filter Chain par défaut

- **UsernamePasswordAuthenticationFilter**
  - analyse une soumission de formulaire d'authentification, qui doit fournir un couple *username*/*password*. Ce filtre est activé par défaut sur l'URL */login*.
- **DefaultLoginPageGeneratingFilter**
  - construit une page d'authentification par défaut, à moins d'être explicitement désactivé. C'est pour cette raison qu'une page de login apparaît lors de l'activation de Spring Security, avant même que le développeur ne code une page personnalisée.
- **DefaultLogoutPageGeneratingFilter**
  - construit une page de déconnexion par défaut, à moins d'être explicitement désactivé.
- **BasicAuthenticationFilter**
  - Vérifie la présence dans la requête reçue d'un *header* de type *basic auth*, et le cas échéant, essaie d'authentifier l'utilisateur avec le couple *username*/*password* récupéré dans ce *header*.
- **RequestCacheAwareFilter :**
  - Vérifie dans le cache des requêtes si une requête passée correspond à la requête en cours pour en accélérer le traitement.

Filter Chain par défaut

- **SecurityContextHolderAwareRequestFilter**
  - agrémente chaque requête d'un *wrapper* offrant diverses fonctionnalités, notamment en ce qui concerne l'authentification : récupération de l'utilisateur, vérification qu'il est bien authentifié, récupération de ses rôles, possibilité d'authentification via l'*Authentication Manager*, possibilité de déconnexion via les *logout handlers*, maintient de cohérence du *Security Context* dans les différents threads…
- **AnonymousAuthenticationFilter**
  - fournit un objet de type *Authentication* au *Security Context Holder* s'il n'en a aucun.
- **SessionManagementFilter**
  - vérifie qu'un utilisateur est authentifié depuis le début de la requête, et effectue le cas échéant des actions relatives à la session, comme la vérification de la présence de plusieurs logins concurrents.
- **ExceptionTranslationFilter**
  - Gère les *AccessDeniedException* et *AuthenticationException* levées par la *filterChain*. Ce filtre est nécessaire car il fait le lien (la traduction) entre les exceptions Java et les réponses *http*, permettant de maintenir la viabilité l'IHM en cas d'erreur.
- **FilterSecurityInterceptor**
  - Effectue les vérifications d'autorisations basées sur les rôles de l'utilisateur connecté.

Authentification

- Il faut distinguer 3 scénarios d'authentification :
  - Les données des utilisateurs (identifiant + mot de passe) sont stockés dans une base de données à laquelle le développeur a accès : c'est le cas le plus courant, et celui qui sera détaillé ici.
  - L'application n'a pas directement accès à ces informations et doit passer par exemple par un service REST tiers pour l'authentification. Ce cas s'applique par exemple dans le cadre d'une utilisation d'Atlassian Crowd. On retiendra qu'il se traite comme le premier cas, à l'exception de cette couche intermédiaire d'interrogation du service d'authentification qu'il faut ajouter.
  - L'authentification est effectuée via OAuth2 (cas d'un "login with Google" par exemple). Ce cas demande nettement plus d'explications et ne sera pas détaillé ici.

Authentification

- En considérant que les mots de passe des utilisateurs sont chriffrés dans la base de donnée , deux beans doivent être déclarés pour que l'authentification soit opérationnelle : une implémentation de l'interface **UserDetailsService**, et un **PasswordEncoder.**
  - UserDetailsService :
    - l'implémentation de cette interface doit comporter une méthode renvoyant un objet de type **UserDetails** à partir d'un simple identifiant d'utilisateur. Cet objet contient à minima le couple *username*/*password*, ainsi que généralement la liste des rôles (c'est à dire les autorisations) de l'utilisateur. Il est tout à fait possible d'utiliser/d'étendre les implémentations toutes faites fournies par Spring Security.
  - PasswordEncoder :
    - permet de spécifier quel algorithme d'encryption utiliser sur les mots de passe. L'algorithme par défaut de Spring Security est *BCrypt*. Il est tout à fait possible d'utiliser différents algorithmes selon les utilisateurs, option sur laquelle nous ne nous attarderons pas.

Authentification

- Mise en place avec BasicAuthentificationFilter

1. Extraction du couple *username*/*password* d'un *header* de la requête, valorisé grâce au formulaire de connexion rempli par l'utilisateur (cette étape est automatique).
2. Utilisation du bean *userDetailsService* pour récupérer (en base de données, à partir du username renseigné par l'utilisateur) les informations nécessaires à la confection d'un objet *UserDetails*. Cet objet contient le mot de passe crypté de l'utilisateur.
3. Encryption automatique du mot de passe renseigné dans le formulaire de connexion, et comparaison avec le mot de passe contenu dans l'objet *UserDetails*. L'utilisateur est authentifié si la comparaison révèle que les mots de passe sont identiques.

Authentification

- Spring Security distingue deux objets relatifs à l'autorisation :
  - Les **Authorities** : dans sa forme la plus basique, une *Authority* n'est qu'une simple chaîne de caractères désignant une responsabilité : "user", "ADMIN", "grandDictateurEnChef"…
  - Les **Roles** : un *Role* n'est ni plus ni moins qu'une *Authority* précédée du préfixe "ROLE_"

Authentification

- Comment dire à Spring Security quelles URLs protéger, et avec quelles restrictions?
- Certaines URLs seront accessibles à tout le monde (connecté ou non), certaines page ne seront visibles que pour les utilisateurs connectés, et certaines pages ne seront de surcroît accessibles qu'aux utilisateurs disposant d'autorisations particulières.
- La classe dans laquelle renseigner tous ces détails répond à 3 critères principaux :
  - elle hérite de **WebSecurityConfigurerAdapter** => elle surcharge (entre autres) la méthode *configure(HttpSecurity)*.
  - elle est annotée ***@Configuration*** => cette annotation indique que le principal but de la classe est la déclaration et l'instanciation de beans (c'est à dire d'objets qui doivent être gérés par Spring).
  - elle est également annotée ***@EnableWebSecurity*** => l'ajout de cette annotation à la précédente est la façon de dire à Spring Security "tire ta configuration de ce **WebSecurityConfigurerAdapter** là".

http de la méthode configure

- **http** : il s'agit de l'objet HttpSecurity donné en paramètre, celui que nous voulons configurer.
- **authorizeRequests** : la suite de la configuration concerne des requêtes vers des URLs, définies (dans cet exemple) par des "*antMatchers*". Un antMatcher permet de désigner un pattern selon les règles principales suivantes : **?** = n'importe quel caractère non nul; ***** = n'importe quel caractère (possiblement nul); ****** = n'importe que nombre de répertoires dans le chemin de l'URL; **{nomVariable:[a-z]+}** = la regex [a-z]+, stockée dans une variable "nomVariable".
- **hasRole("ADMIN")** : l'URL est accessible à un utilisateur authentifié, qui a le rôle *ADMIN*, correspondant à la chaîne de caractères "*ROLE_ADMIN*". Le contrôle *hasRole("ADMIN")* est strictement équivalent au contrôle *hasAuthority("ROLE_ADMIN")*.
- **hasAuthority** : l'URL est accessible à un utilisateur authentifié, qui bénéficie de l'autorisation *PRESQUE_ADMIN*, correspondant à la chaîne de caractères "*PRESQUE_ADMIN*".
- **hasAnyAuthority** : l'URL est accessible à un utilisateur authentifié, qui bénéficie d'au moins l'une des 2 autorisations suivantes : *PRESQUE_ADMIN* (chaîne de caractères "*PRESQUE_ADMIN*"), ou *ROLE_ADMIN* (chaîne de caractères "*ROLE_ADMIN*"). Le contrôle *hasAuthority("ROLE_ADMIN")* est strictement équivalent au contrôle *hasRole("ADMIN")*.

http de la méthode configure

- **authenticated** : l'URL est accessible à n'importe quel utilisateur authentifié, indépendamment de ses rôles/autorisations.
- **permitAll** : l'URL est accessible sans authentification nécessaire (souvent, seule la page d'authentification est accessible à tout le monde).
- **anyRequest** : définit le comportement par défaut de toute URL non précisée précédemment.
- **and** : la configuration des requêtes est terminée, mais nous souhaitons configurer d'autres aspects de Spring Security (voir points **10** et **11**).
- **formLogin** : l'authentification est autorisée via le formulaire de login.
- **httpBasic** : l'authentification est autorisée via un header *BasicAuth*.

```
.antMatchers( ...antPatterns: "/admin/{adminName}/**") ExpressionUrlAuthorizationConfigurer<H>.AuthorizedUrl
    .access( attribute: "hasRole('ADMIN') and hasIpAddress('192.168.1.0/24') and @monBeanPerso.checkAdminName(#adminName)")
```

Defense in depth

- Contrôler l'accès aux méthodes que l'on peut trouver dans les @Controllers, les @Components, les @Services, les @Repositories, et autres beans Spring.
- Cette approche est mise en place par le biais d'annotations affectant les méthodes publiques des beans.
- Pour rendre possible cette sécurité supplémentaire, il est nécessaire de l'autoriser explicitement dans la classe annotée *@Configuration* que nous avons survolée dans la partie précédente, en ajoutant l'annotation **@EnableGlobalMethodSecurity** :

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(
        prePostEnabled = true,
        securedEnabled = true,
        jsr250Enabled = true)
public class WebSecurityConfig  extends WebSecurityConfigurerAdapter {
```

Defense in depth

- **prePostEnabled** : permet d'utiliser les annotations *@PreAuthorize* et *@PostAuthorize*
- **securedEnabled** : permet d'utiliser l'annotation *@Secured*
- **jsr250Enabled** : permet d'utiliser l'annotation *@RolesAllowed*.

```
@Secured("ROLE_ADMIN")
public void methodExample(String param){
    // some magnificient code
}
```

```
@RolesAllowed("ADMIN")
public void methodExample(String param){
    // some magnificient code
}
```

```
@RolesAllowed("ADMIN")
public void methodExample(String param){
    // some magnificient code
}
```

Defense in depth

- Les annotation *@PreAuthorize et @PostAuthorize* sont considérées comme plus puissantes que les précédentes, car elles peuvent accepter n'importe quelle expression SpEL aussi bien qu'un rôle ou une autorisation. *@PreAuthorize* effectue le contrôle avant d'entrer dans la méthode, tandis que *@PostAuthorize* l'effectue après l'exécution de la méthode, ayant la possibilité d'en modifier le résultat.

Protection CSRF

- Certaines requêtes modifient l'état de l'application : elle correspondent généralement aux mot-clefs POST, PUT, DELETE et parfois OPTIONS. Ces requêtes sont parfois soumises à l'application, en provenance d'une page qui n'a en apparence rien à voir, de façon malveillante et imperceptible pour l'utilisateur. Afin d'éviter ce genre d'attaque, il est possible d'utiliser le CsrfFilter de Spring Security (voir la partie sur la Filter Chain) en ajoutant la méthode *csrf()* à la configuration de la sécurité web :

```
@Override
protected void configure(HttpSecurity http) throws Exception{
    http
            .csrf() CsrfConfigurer<HttpSecurity>
            .and() HttpSecurity
            .authorizeRequests().antMatchers( ...antPatterns: "/api/
```

Configuration simple

- Uniquement les utilisateurs authentifiés

```
@Configuration
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
	@Override
	protected void configure(HttpSecurity http) throws Exception {
		http .authorizeRequests() .
			antMatchers("/", "/home").permitAll() .
			anyRequest().authenticated() .and()
			.formLogin() .loginPage("/login") .permitAll() .and()
			.logout() .permitAll();
	}
	@Bean
	@Override
	public UserDetailsService userDetailsService() {
		UserDetails user = User.withDefaultPasswordEncoder()
			.username("user") .password("password") .roles("USER") .build();
		return new InMemoryUserDetailsManager(user);
	}
}
```

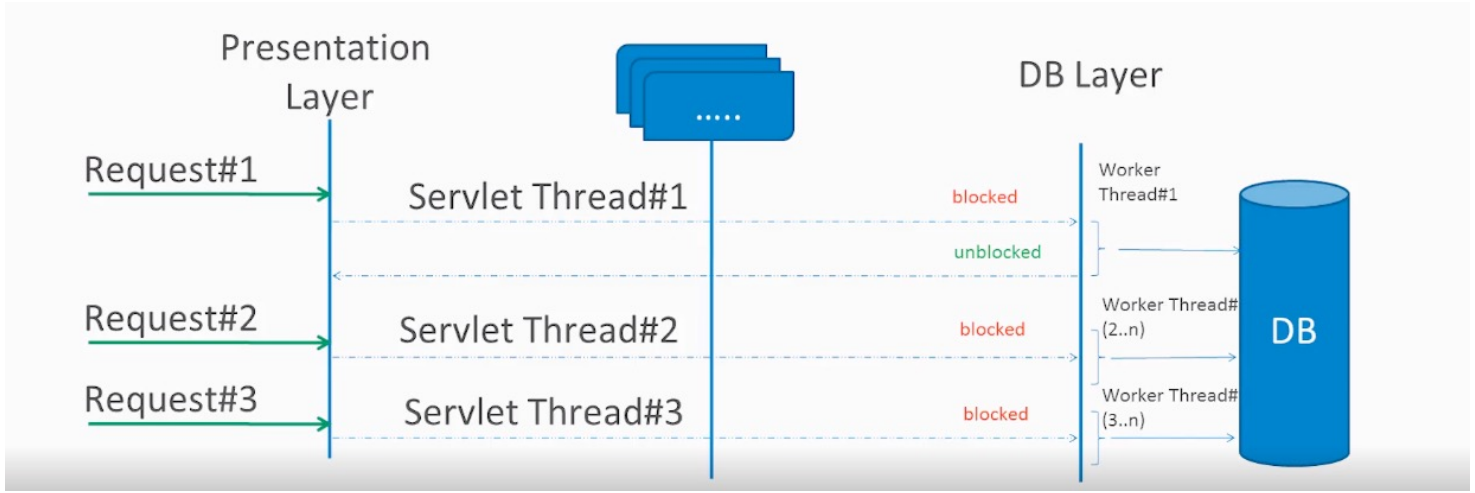
- Des questions?

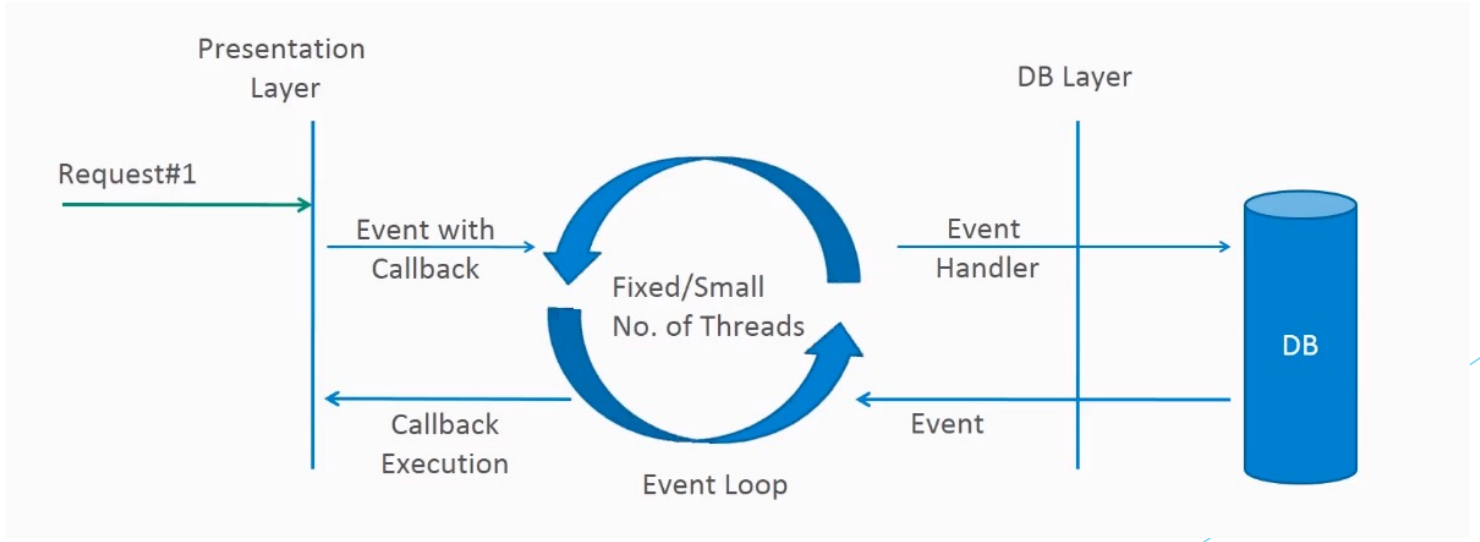
Webflux

Programmation réactive

- La programmation réactive est un paradigme de programmation qui favorise une approche asynchrone, non bloquante et événementielle du traitement des données.
- La programmation réactive implique la modélisation des données et des événements en tant que flux de données observables et la mise en œuvre de routines de traitement des données pour réagir aux changements dans ces flux.
- Le terme « réactif » fait référence à des modèles de programmation construits autour de la réaction aux changements. Il est construit autour d'un modèle éditeur-abonné (modèle observateur).
- Dans un style de programmation réactif, nous faisons une demande de ressource et commençons à effectuer d'autres choses.
- Lorsque les données sont disponibles, nous recevons la notification ainsi que les données informant de la fonction de rappel.
- Dans la fonction de rappel, nous gérons la réponse selon les besoins de l'application/utilisateur. Une chose importante à retenir est la contre-pression.
- En code non bloquant, il devient important de contrôler le taux d'événements afin qu'un producteur rapide ne submerge pas sa destination.
- La programmation Web réactive est idéale pour les applications qui ont des données en streaming et les clients qui les consomment et les diffusent à leurs utilisateurs.
- Ce n'est pas idéal pour développer des applications CRUD traditionnelles. Si vous développez le prochain Facebook ou Twitter avec beaucoup de données, une API réactive pourrait être exactement ce que vous recherchez.

Sync vs async





Côté server

- Même principe de contrôleurs
- Principale différence : HandlerMapping, HandlerAdapter, ne sont pas bloquants et fonctionnent sur les ServerHttpRequest et ServerHttpResponse réactifs plutôt que sur ls HttpServletRequest et HttpServletResponse.

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

HandlerFunctions

- Les requêtes HTTP entrantes sont gérées par une HandlerFunction, qui est essentiellement une fonction qui prend une ServerRequest et renvoie une Mono<ServerResponse>.
- ServerRequest et ServerResponse sont des interfaces immuables qui offrent un accès convivial JDK-8 aux messages HTTP.
- Les deux sont entièrement réactifs en s'appuyant sur Reactor : la demande expose le corps en tant que Flux ou Mono ; la réponse accepte n'importe quel éditeur Reactive Streams comme corps.
- ServerRequest donne accès à divers éléments de requête HTTP : la méthode, l'URI, les paramètres de requête et — via l'interface distincte ServerRequest.Headers — les en-têtes. L'accès au corps est assuré par les méthodes corporelles.
- Par exemple, voici comment extraire le corps de la requête dans un Mono<String> :
  - Mono<String> string = request.bodyToMono(String.class);

Mono

- L'API Mono vous permet d'émettre une seule valeur, après quoi elle se terminera immédiatement. Cela signifie que le Mono est la contrepartie réactive du retour d'un objet simple, ou d'un Optionnel.

```
public Optional<Person> findCurrentUser() {
    if (isAuthenticated())
        return Optional.of(new Person("Jane", "Doe"));
    else return Optional.empty(); }
```

Devient en reactive

```
public Mono<Person> findCurrentUser() {
    if (isAuthenticated())
        return Mono.just(new Person("Jane", "Doe"));
    else return Mono.empty(); }
```

Flux

- Alors que le Mono est utilisé pour gérer zéro ou un résultat, le Flux est utilisé pour gérer de zéro à plusieurs résultats, peut-être même des résultats infinis.
- Remplace le Stream en vulgarisation

```
public Stream<Person> findAll() {
    return Stream.of( new Person("Jane", "Doe"), new Person("John", "Doe") ); }
```

- devient

```
public Flux<Person> findAll() {
    return Flux.just( new Person("Jane", "Doe"), new Person("John", "Doe") ); }
```

Avec @Component

Classe qui ne retourne qu'une liste d'objets.

```
@Component
public class ProductHandler {

private List<Product> products = Arrays.asList(
    new Product(1, "Lit", "ADB123"),
    new Product(2, "Canapé", "ADPO23"),
    new Product(3, "Chaise", "OPMLE23")
);

public Mono<ServerResponse> getProduct(ServerRequest serverRequest) {
    return ServerResponse.ok().

        contentType(MediaType.APPLICATION_JSON).
        body(BodyInserters.fromObject(products));
}

}
```

Ajouter un router pour les routes

```
@Configuration
public class ProductRouter {

    @Bean
    public RouterFunction<ServerResponse> route(ProductHandler productHandler){
        return RouterFunctions.route(RequestPredicates.GET("/products")
        .and(RequestPredicates.accept(MediaType.APPLICATION_JSON)),
        productHandler::getProduct);
    }
}
```

Côté client

```
WebClient client = WebClient.create("http://example.com");
Mono<Account> account = client.get()
    .url("/accounts/{id}", 1L)
    .accept(APPLICATION_JSON) .exchange(request)
    .flatMap(response ->
        response.bodyToMono(Account.class));
```

Dans Spring Boot

```
package fr.tuto.app.reactive.rest;

import fr.tuto.app.reactive.rest.client.ProductWebClient;
import org.springframework.boot.SpringApplication;
import
org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class RestApplication {

    public static void main(String[] args) {

        SpringApplication.run(RestApplication.class, args);
        ProductWebClient gwc = new ProductWebClient();
        System.out.println(gwc.getResult());
    }
}
```

- Des questions?

Introduction aux modules Java 9

Module

- Java 9 introduit un nouveau niveau d'abstraction au-dessus des packages, officiellement connu sous le nom de Java Platform Module System (JPMS), ou « Modules » en abrégé.
- Un module est un groupe de packages et de ressources étroitement liés ainsi qu'un nouveau fichier descripteur de module.
- En d'autres termes, il s'agit d'une abstraction « package of Java Packages » qui nous permet de rendre notre code encore plus réutilisable.

Package

- Les packages à l'intérieur d'un module sont identiques aux packages Java que nous utilisons depuis la création de Java.
- Lorsque nous créons un module, nous organisons le code en interne dans des packages comme nous le faisions auparavant avec n'importe quel autre projet.
- En plus d'organiser notre code, les packages sont utilisés pour déterminer quel code est accessible publiquement en dehors du module.

Ressources

- Chaque module est responsable de ses ressources, comme les médias ou les fichiers de configuration.
- Auparavant, nous placions toutes les ressources au niveau racine de notre projet et gérons manuellement quelles ressources appartenaient aux différentes parties de l'application.
- Avec les modules, nous pouvons expédier les images et les fichiers XML requis avec le module qui en a besoin, ce qui rend nos projets beaucoup plus faciles à gérer.

Descriptor

- Lorsque nous créons un module, nous incluons un fichier descripteur qui définit plusieurs aspects de notre nouveau module : Nom – le nom de notre module Dépendances - une liste d'autres modules dont ce module dépend Packages publics - une liste de tous les packages que nous voulons accessibles depuis l'extérieur du module Services offerts - nous pouvons fournir des implémentations de services qui peuvent être utilisées par d'autres modules Services consommés - permet au module actuel d'être un consommateur d'un service Autorisations de réflexion - permet explicitement à d'autres classes d'utiliser la réflexion pour accéder aux membres privés d'un package Les règles de nommage des modules sont similaires à la façon dont nous nommons les packages (les points sont autorisés, les tirets ne le sont pas). Il est très courant de faire des noms de style de type projet (my.module) ou Reverse-DNS (com.baeldung.mymodule). Nous utiliserons le style de projet dans ce guide. Nous devons répertorier tous les packages que nous voulons rendre publics car par défaut, tous les packages sont privés de module. Il en est de même pour la réflexion. Par défaut, nous ne pouvons pas utiliser la réflexion sur les classes que nous importons d'un autre module.

Types de modules

- Modules système - Ce sont les modules répertoriés lorsque nous exécutons la commande list-modules ci-dessus. Ils incluent les modules Java SE et JDK.
- Modules d'application - Ces modules sont ce que nous voulons généralement construire lorsque nous décidons d'utiliser des modules. Ils sont nommés et définis dans le fichier module-info.class compilé inclus dans le JAR assemblé.
- Modules automatiques – Nous pouvons inclure des modules non officiels en ajoutant des fichiers JAR existants au chemin du module. Le nom du module sera dérivé du nom du JAR. Les modules automatiques auront un accès en lecture complet à tous les autres modules chargés par le chemin.
- Module sans nom – Lorsqu'une classe ou un JAR est chargé sur le chemin de classe, mais pas le chemin du module, il est automatiquement ajouté au module sans nom. C'est un module fourre-tout pour maintenir la compatibilité descendante avec le code Java précédemment écrit.

Déclartation

```
module myModuleName {
    // all directives are optional
}
```

Requires pour les imports

- Des questions?
- Merci pour votre attention.