Spring - Une introduction

NSY102 - 2020

Stéphane Bruyère

le cnam

Spring - introduction

Spring est un framework open source pour le développement d'applications java, qui n'est pas standardisé par le JCP (java community process qui émet les JSR) ...

... mais est un standard de facto du fait de sa large utilisation.

Il permet d'intégrer les framework open source (Hibernate ...) et les principales API java (Servlet, JMS ...)

Spring peut s'utiliser dans un simple conteneur web (Tomcat) ...

... car il utilise de simples POJO pour développer des applications plutôt que d'utiliser des EJB dans un conteneur.

Spring - introduction

Le noyau de Spring est basé sur une fabrique générique de composants informatiques (beans) et un conteneur capable de stocker ces beans.

De plus, le noyau de Spring permet de forcer le contrôle de ces composants de leur extérieur, par l'inversion de contrôle.

Le principal avantage est de composer les beans de façon déclarative plutôt que de façon impérative dans le programme

source wikipedia

Spring - l'inversion de contrôle

Spring s'appuie donc sur l'inversion de contrôle, assurée de deux façons différentes : la recherche de dépendances et l'injection de dépendances.

Pour mémoire (cf cours NFP121), l'inversion de contrôle fournit un moyen cohérent de configuration et de gestion des objets Java à l'aide de l'introspection.

L'injection de dépendances est un mécanisme qui permet d'implémenter le principe de l'inversion de contrôle.
Il consiste à créer dynamiquement (injecter) les dépendances entre les différents objets en s'appuyant sur une description (fichier de configuration ou métadonnées) ou de manière programmatique. Ainsi les dépendances entre composants logiciels ne sont plus exprimées dans le code de manière statique mais déterminées dynamiquement à l'exécution.

Source Wikipedia

Spring - l'inversion de contrôle

- **La recherche de dépendance consiste à interroger le conteneur, afin d'obtenir un objet avec un nom spécifique ou d'un type spécifique. C'est un cas de fonctionnement similaire aux EJBs.**
- **L'injection de dépendances - cette injection peut être effectuée de trois manières possibles :**
  - L'injection de dépendance via le constructeur.
  - L'injection de dépendance via les modificateurs (setters).
  - L'injection de dépendance via une interface.

Les deux premières sont les plus utilisées par Spring.

source wikipedia

Spring - l’inversion de contrôle

Avec Spring, on ne crée pas directement d'objet, mais on décrit comment un objet doit être créé. Le framework se charge de la création.

De même, les services et les composants ne sont pas appelés directement. Les services et les composants devant être appelés sont définis dans un fichier de configuration .

L’inversion de contrôle est destinée à augmenter la facilité de maintenance et de test.

Spring - l’inversion de contrôle

Les annotations de Spring permettent de marquer des classes comme beans, afin qu’ils soit reconnus comme tels par le container et puissent être « injectés ».

Avec les annotations @Component, @Repository, @Service et @Controller et l'analyse automatique des composants activée, Spring importera automatiquement les beans dans le conteneur et les « injectera » quand nécessaire.

L’annotation @Autowired gère le câblage. Elle est à placer au-dessus de l’attribut à récupérer par injection dans une classe. Cet objet devra avoir été annoté par l’une des annotations vues au-dessus.

Spring - l’inversion de contrôle

L'annotation @Component marque une classe java comme un bean afin que le mécanisme d'analyse des composants de Spring puisse la repérer et l'intégrer au contexte de l'application.

Les autres annotations sont des spécialisations de celle-ci pour les DAO (@Repository), les services -« Business Service Facade »- (@Service) et les contrôleurs (servlets en JEE) (@Controller).

Elles permettent de préciser l’intention, et peuvent amener des fonctionnalités supplémentaires, comme la gestion des exceptions par DataAccessException pour @Repository, ou la possibilité d’utiliser l’annotation additionnelle @RequestMapping pour @Controller.

Spring - programmation orientée aspect

Ce type de programmation (AOP) permet de prendre en compte les fonctionnalités transverses.

Par exemple, une couche logicielle dédiée à gérer la logique métier applicative, va se retrouver dépendante de modules gérant les aspects transactionnels, de journalisation, etc. ; la croissance de ces dépendances conduit à une complexification du code, de son développement et de sa maintenance. La programmation par aspect va permettre d'extraire les dépendances entre modules concernant des aspects techniques entrecroisés et de les gérer depuis l'extérieur de ces modules en les spécifiant dans des composants du système à développer nommés « aspects » ; ils sont développés à un autre niveau d'abstraction *(source wikipedia)*.

Spring - programmation orientée aspect

De façon schématique, la programmation orientée aspect permet d'ajouter automatiquement du code avant et après certains appels de méthodes.

L'AOP permet donc d'écrire le code sans se soucier des préoccupations transverses en les ajoutant et les traitant systématiquement et de manière déclarative.

Spring va utiliser les concepts de l'AOP notamment pour la mise en œuvre des transactions, les logs, la gestion des droits d'accès ...

source S. Rosmorduc

Spring - la couche d'abstraction

La couche d'abstraction de Spring permet d'intégrer d'autres frameworks et bibliothèques avec une plus grande facilité.

Grâce à cette couche, Spring ne concurrence pas d'autres frameworks dans une couche spécifique d'un modèle architectural MVC mais s'avère être un framework multi-couches pouvant s'insérer au niveau de toutes les couches : modèle, vue et contrôleur.

Ainsi il permet d'intégrer Hibernate pour la couche de persistance ou encore JavaServer Faces (JSF)pour la couche présentation.

source wikipedia

Spring - modules

Spring fournit un ecosystème particulièrement riche avec plus de 50 sous-projets :

Spring Boot : Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run".

Spring Data : Spring Data's mission is to provide a familiar and consistent, Spring-based programming model for data access while still retaining the special traits of the underlying data store.

Spring MVC : Spring Web MVC is the original web framework built on the Servlet API and has been included in the Spring Framework from the very beginning.

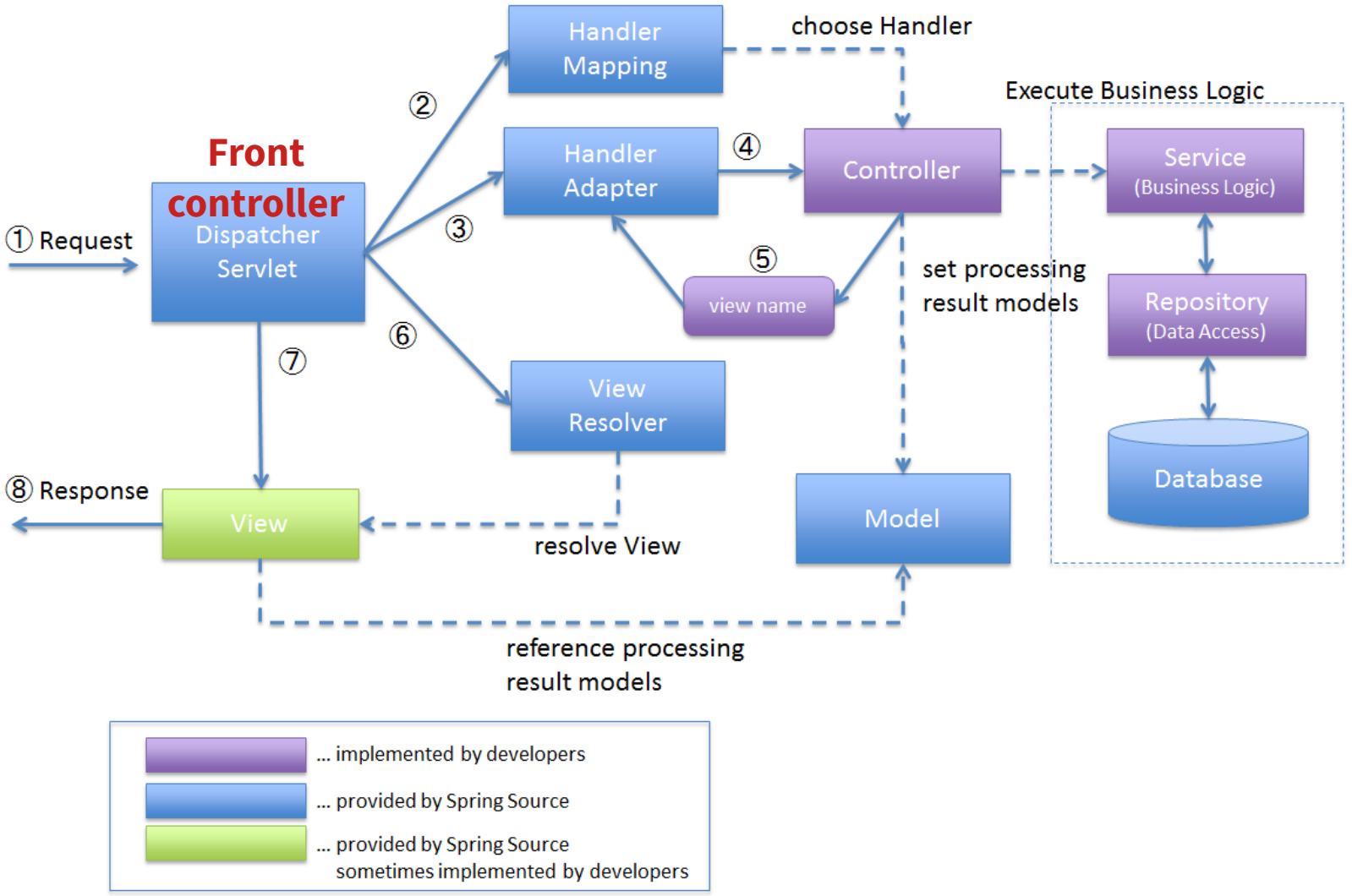
Spring Security, Spring Websocket, Spring AMPQ, Spring Web Services, Spring Cloud, Spring Mobile ...

Spring - Spring Boot

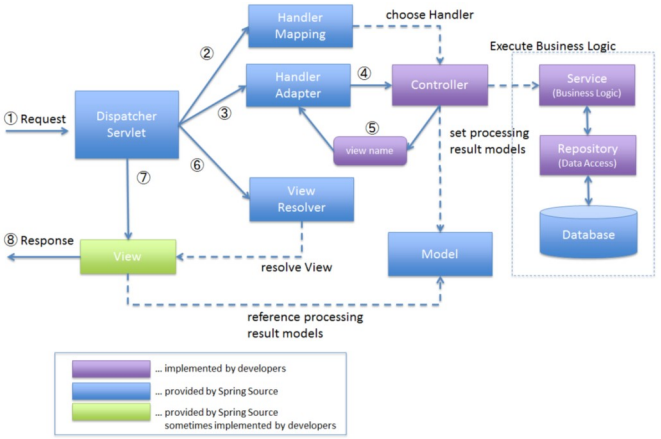
La configuration de Spring est particulièrement compliquée.
Spring Boot a donc été créé dans le but de faciliter la configuration d'un projet Spring et réduire le temps alloué à son démarrage.
Pour cela Spring Boot propose :

- Un site web (https://start.spring.io/) permettant de générer rapidement la structure d'un projet en y incluant toutes les dépendances Maven nécessaires.
- des « Starters » pour gérer les dépendances. Les dépendances Maven sont regroupées dans des « méga dépendances » afin de faciliter leur gestion.
- L'auto-configuration, qui applique une configuration par défaut au démarrage de l'application pour toutes dépendances présentes . Cette configuration s'active à partir du moment où l'application est annotée avec « @SpringBootApplication » (cette configuration peut-être surchargée). L'auto-configuration simplifie la configuration sans pour autant vous restreindre dans les fonctionnalités de Spring.
- possibilité d'intégrer directement un serveur Tomcat dans l'exécutable. Un Tomcat embarqué sera démarré au lancement afin de faire tourner l'application.

Spring - architecture modèle web



Spring - architecture modèle web



1- DispatcherServlet receives the request.

2- DispatcherServlet dispatches the task of selecting an appropriate controller to HandlerMapping. HandlerMapping selects the controller which is mapped to the incoming request URL and returns the (selected Handler) and Controller to DispatcherServlet.

3- DispatcherServlet dispatches the task of executing of business logic of Controller to HandlerAdapter.

4- HandlerAdapter calls the business logic process of Controller.

5- Controller executes the business logic, sets the processing result in Model and returns the logical name of view to HandlerAdapter.

6- DispatcherServlet dispatches the task of resolving the View corresponding to the View name to ViewResolver. ViewResolver returns the View mapped to View name.

7- DispatcherServlet dispatches the rendering process to returned View.

8- View renders Model data and returns the response.

Spring - Le POM Spring Boot

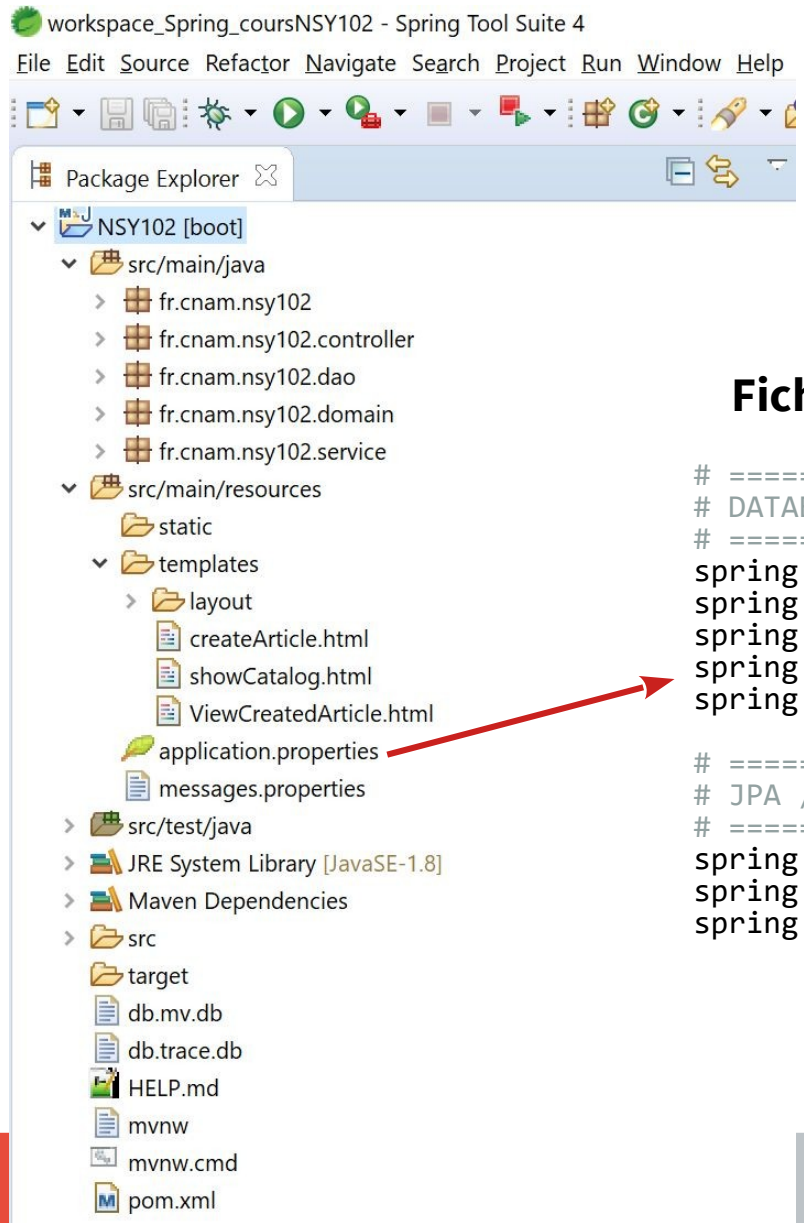
```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
	<modelVersion>4.0.0</modelVersion>
	<parent>
		<groupId>org.springframework.boot</groupId>
		<artifactId>spring-boot-starter-parent</artifactId>
		<version>2.3.0.RELEASE</version>
		<relativePath/> <!-- lookup parent from repository -->
	</parent>
	<groupId>fr.cnam</groupId>
	<artifactId>NSY102</artifactId>
	<version>0.0.1-SNAPSHOT</version>
	<name>NSY102</name>
	<description>Demo project for Spring Boot</description>

	<properties>
		<java.version>11</java.version>
	</properties>

	<dependencies>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-data-jpa</artifactId>
		</dependency>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-thymeleaf</artifactId>
		</dependency>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-web</artifactId>
		</dependency>

		<dependency>
			<groupId>com.h2database</groupId>
			<artifactId>h2</artifactId>
			<scope>runtime</scope>
		</dependency>
```

généré via https://start.spring.io/
ou via Eclipse
pour les éléments pricipaux
(web, jpa, Thymeleaf, h2)

Spring - Le POM Spring Boot

```
...
<dependency>
<groupId>org.webjars</groupId>
<artifactId>bootstrap</artifactId>
<version>4.5.0</version>
</dependency>
<dependency>
<groupId>org.webjars</groupId>
<artifactId>jquery</artifactId>
<version>3.5.1</version>
</dependency>

<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-test</artifactId>
<scope>test</scope>
<exclusions>
<exclusion>
<groupId>org.junit.vintage</groupId>
<artifactId>junit-vintage-engine</artifactId>
</exclusion>
</exclusions>
</dependency>
</dependencies>

<build>
<plugins>
<plugin>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-maven-plugin</artifactId>
</plugin>
</plugins>
</build>

</project>
```

Spring - arborescence Spring Boot



Fichier de config application.properties

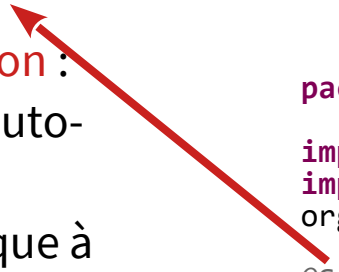
```
# ===============================
# DATABASE
# ===============================
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.url=jdbc:h2:./db
spring.datasource.username=user
spring.datasource.password=user
spring.h2.console.enabled=true

# ===============================
# JPA / HIBERNATE
# ===============================
spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.H2Dialect
```

Spring - le fichier de lancement de Spring Boot

Généré par Spring boot

regroupe les annotations :

- @EnableAutoConfiguration : permet le mécanisme d'auto-configuration
- @ComponentScan : indique à Spring dans quels packages rechercher les beans (classes annotées)
- @Configuration : Les classes où elle est apposée se substituent aux traditionnels fichiers de configuration XML. On y trouve la déclaration de beans Spring, l'import de fichiers ou de classes de configuration.



```
package fr.cnam.nsy102;

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

@SpringBootApplication
public class Nsy102Application {

	public static void main(String[] args) {
		SpringApplication.run(Nsy102Application.class, args);
	}
}
```

Spring - un 1er exemple

un première appli web « Hello World ! »

annotation @Controller, le contrôleur MVC
<=> servlet en JEE

```
@Controller
public class SimpleController {

@RequestMapping("/simple")
@ResponseBody
public String simple() {
return "Hello World !";
}

}
```

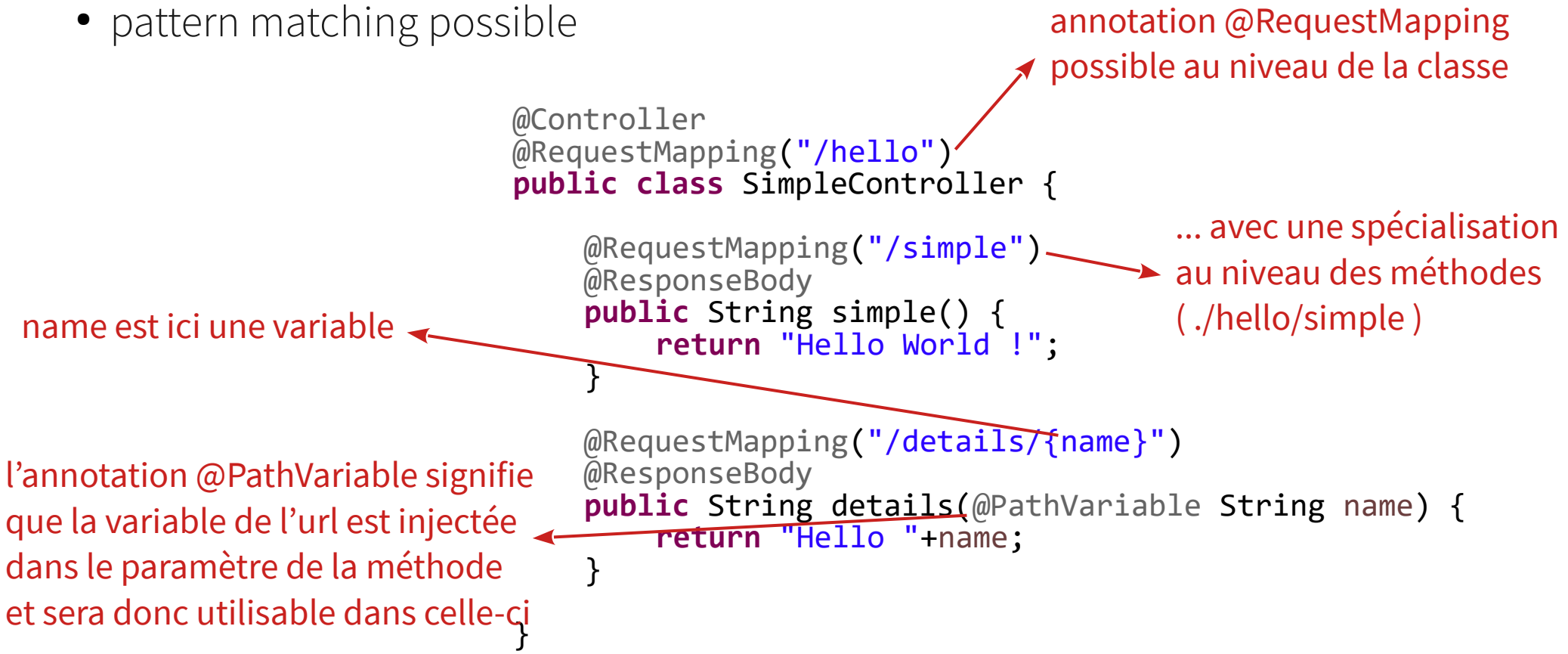
annotation @RequestMapping,
donne l'URL par laquelle on
pourra invoquer l'action

annotation @ResponseBody
signifie que le résultat s'affichera
dans le corps de la page html

Spring - un 2ème exemple

url paramétrée avec @PathVariable

- pattern matching possible

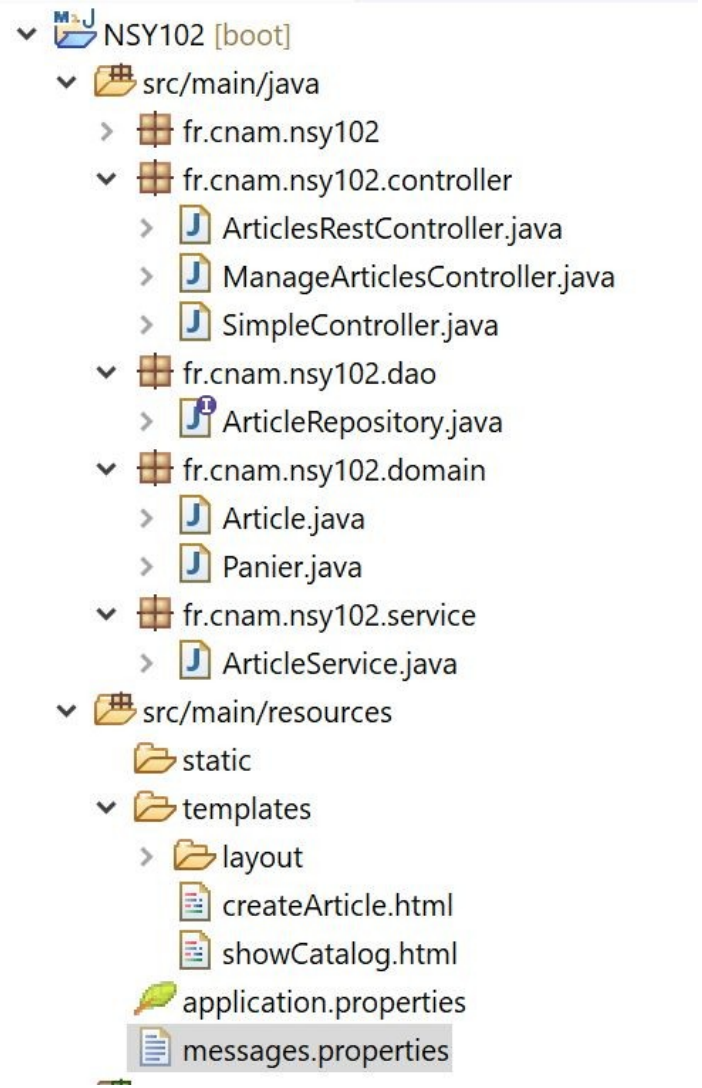


Spring - Femto Application

Nous allons construire une application très basique, qui comprend une seule classe métier, « article ».

Nous devrons pouvoir créer des articles, les afficher, les supprimer, les mettre dans notre panier (dans le cadre d'une session).

Classement de nos classes dans l'arborescence selon leur typologie (métier, contrôleur, service, dao)



Spring - Application : classe métier

prise en charge par JPA

clé primaire

```
@Entity
public class Article {

    @Id
    private String id;

    private String name;

    private double price;

    public String getId() {
        return id;
    }
    public void setId(String id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public double getPrice() {
        return price;
    }
    public void setPrice(double price) {
        this.price = price;
    }
}
```

Compte tenu de notre configuration (`spring.jpa.hibernate.ddl-auto=update`), Spring créera la db si elle n'existe pas au lancement du programme.

Spring - Application : classe métier

```
public class Panier {

    private List<Article> articles;

    public Panier() {
        articles = new ArrayList<>();
    }

    public List<Article> getArticles() {
        return this.articles;
    }

    public void setArticles(List<Article> articles) {
        this.articles = articles;
    }
}
```

Le panier, que nous ne persisterons pas

Spring - Application : DAO

Interfaces Spring xxxRepository, spécifiques pour l'accès aux données via JPA.

Les repositories génériques fournis par Spring permettent de réaliser les opérations courantes d'accès aux données.

On trouve notamment les Interfaces Repository, CrudRepository ...

Il suffit de créer une interface héritant d'un de ces repositories pour utiliser ses méthodes, et en définir de plus spécifiques si nécessaire (en respectant les conventions de nommage, par ex *findByAttribut*).

2 paramètres génériques : le type de l'entité et le type de sa clé primaire.

Le code sera généré par Spring Data à partir de notre interface, sans que nous ayons à fournir nous mêmes l'implémentation !

```
public interface ArticleRepository
extends JpaRepository<Article, String> {}
```

Spring - application : business service facade

```java
@Service
public class ArticleServiceImpl implements ArticleService {

    @Autowired
    private ArticleRepository articleRepository;

    @Transactional
    public Article enregistrer(Article article) {
        return articleRepository.save(article);
    }

    @Transactional
    public void supprimer(String id) {
        articleRepository.deleteById(id);
    }

    public List<Article> lister() {
        return articleRepository.findAll();
    }

    public Optional<Article> trouver(String id) {
        return articleRepository.findById(id);
    }

}
```

composant / bean de type service.
L'annotation permettra au bean d'être importé dans le conteneur et injectable à la demande

injection du bean DAO

méthode transactionnelle (cf slide suivante)

```java
public interface ArticleService {

public Article enregistrer(Article article) ;
public void supprimer(String id) ;
public List<Article> lister();
public Optional<Article> trouver(String id) ;
}
```

Spring - Les transactions avec Spring Boot

La gestion des transactions est configurée par défaut avec Spring Boot, pour autant qu'une dépendance de type « spring-data-* » soit dans le classpath.

L'annotation *@Transactional* au niveau d'une classe ou d'une méthode (forcément *public*) suffira à la rendre transactionnelle.

L'annotation prend également en charge les configurations suivantes (entre autres) :

- le type de propagation de la transaction (la propagation décrit ce qui se passe si une transaction doit être ouverte dans le cadre d'une transaction déjà existante).
- le niveau d'isolement de la transaction (l'isolement détermine entre autres si une transaction peut voir des écritures non validées d'une autre transaction)
- un indicateur readOnly (indicateur pour le fournisseur de persistance que la transaction doit être en lecture seule)
- les règles de roll-back pour la transaction

Spring - Application : un contrôleur 1/2

```java
@Controller
public class ManageArticlesController {

	@Autowired
	ArticleService articleService;

	@GetMapping("/show")
	public String viewTemplate(Model model) {
		List<Article> catalog = articleService.lister();
		model.addAttribute("catalog", catalog);
		return "showCatalog";
	}

	@GetMapping("/new")
	public String createArticle(Model model) {
		model.addAttribute("article", new Article());
		return "createArticle";
	}

	@GetMapping("/delete/{id}")
	public String deleteArticle(@PathVariable String id) {
		articleService.supprimer(id);
		return "redirect:/show";
	}

	@PostMapping("/new")
	public String newArticle(@ModelAttribute Article article, Model model) {
		if (article.getId() != "" && article.getId() != null && article.getName() != "" && article.getName() != null
				&& article.getPrice() != 0.0) {
			articleService.enregistrer(article);
		} else {
			model.addAttribute("error", true);
			return "createArticle";
		}
		return "redirect:/show";
	}
```

- composant / bean de type contrôleur.
- injection du bean Service
- annotation possible dans un bean de type Controller (= @RequestMapping + méthode HTTP GET)
- appelle ShowCatalog.html
- Model = objet permettant de transmettre des paramètres à la vue
- paramètre de l'URL transmis à la méthode
- ModelAttribute = indique que l'argument doit être extrait du modèle. Ici, article est rempli avec les données du formulaire transmis
- redirige vers l'URL /show, càd appelle la méthode ViewTemplate de ce contrôleur

Spring - Application : un contrôleur 2/2

```
@GetMapping("/addToBasket/{id}")
public String addToBasket(HttpSession session, @PathVariable String id) {
      Double totalPrice = 0.0;
      Panier monPanier = (Panier) session.getAttribute("panier");
      if (monPanier == null) {
            monPanier = new Panier();
            session.setAttribute("panier", monPanier);
      }
      Article art = articleService.trouver(id).get();
      monPanier.getArticles().add(art);
      for (Article a : monPanier.getArticles()) {
            totalPrice += a.getPrice();
      }
      session.setAttribute("totalPrice", totalPrice);
      session.removeAttribute("emptyBasket");
      return "redirect:/show";
}

@GetMapping("/removeFromBasket/{index}")
public String removeFromBasket(HttpSession session, @PathVariable int index) {
      Double totalPrice = 0.0;
      ((Panier) session.getAttribute("panier")).getArticles().remove(index);
      Panier monPanier = (Panier) session.getAttribute("panier");
      for (Article a : monPanier.getArticles()) {
            totalPrice += a.getPrice();
      }
      session.setAttribute("totalPrice", totalPrice);
      if (totalPrice == 0)
            session.setAttribute("emptyBasket", "yes");
      else
            session.removeAttribute("emptyBasket");
      return "redirect:/show";
}
}
```

injection de l'objet session ...

... auquel on ajoute des attributs qui survivront tout au long du cycle de vie de l'objet

Spring - Application : un contrôleur REST

```
@RestController
public class ArticlesRestController {

	@Autowired
	ArticleService articleService;

	@GetMapping(path="/showJson", produces = MediaType.APPLICATION_JSON_VALUE )
	public List<Article> viewJson(Model model) {
		List<Article> catalog=articleService.lister();
		return catalog;
	}

	@GetMapping(path="/showJson/{id}", produces = MediaType.APPLICATION_JSON_VALUE )
	public Article viewSingleJson(@PathVariable String id) {
		Article article = articleService.trouver(id).get();
		return article;
	}
}
```

annotation qui combine @Controller et @ResponseBody, et nous évite ainsi de devoir annoter chaque méthode de la classe avec l'annotation @ResponseBody

« produces » donne le type du contenu produit par la méthode (type JSON ici)

Spring - Application : default.html

```
<!DOCTYPE html>
<html lang="fr" xmlns:th="http://www.thymeleaf.org">
<head th:fragment="headerfiles">
<meta content="text/html; charset=UTF-8" http-equiv="Content-Type">
<meta name="viewport"
      content="width=device-width, initial-scale=1, maximum-scale=1.0, user-scalable=no" />
<title>NSY102</title>
<link rel="stylesheet" href="/webjars/bootstrap/4.5.0/css/bootstrap.min.css" media="screen,projection" />
</head>
<body>
      <!-- Fixed navbar -->
      <nav th:fragment="navbar" class="navbar navbar-expand-lg navbar-light bg-light fixed-top">
            <a class="navbar-brand" href="#">NSY102</a>
      </nav>
      <!-- bottom navbar -->
      <nav th:fragment="footernavbar" class="navbar navbar-expand-lg navbar-light bg-light fixed-bottom">
            <a class="navbar-brand" href="#">Cnam 2019 </a>
      </nav>
      <div th:fragment="scripts">
            <script src="/webjars/jquery/3.5.1/jquery.min.js"></script>
            <script src="/webjars/bootstrap/4.5.0/js/bootstrap.min.js"></script>
      </div>
</body>
</html>
```

Thymeleaf est un moteur de template, sous licence Apache 2.0, écrit en Java pouvant générer du XML/XHTML/HTML5. Thymeleaf peut être utilisé dans un environnement web (utilisant l'API Servlet) ou non web. Son but principal est d'être utilisé dans un environnement web pour la génération de vue pour les applications web basées sur le modèle MVC.

source : Wikipedia

Spring - Application : createArticle.html

```
<!DOCTYPE HTML>
<html lang="fr" xmlns:th="http://www.thymeleaf.org">
<head>
<th:block th:insert="layout/default.html :: headerfiles"></th:block>
</head>
<body>
	<!-- <div layout:fragment="content"> -->
	<nav><th:block th:insert="layout/default.html :: navbar"></th:block></nav>
		<div class="container">
			<br> <br> <br>
			<div id="errorMessage" th:if="${error}">
				<h4 class="alert alert-danger" th:text="#{article.error}"></h4>
			</div>
			<form class="form-horizontal" method="post" th:action="@{/new}" th:object="${article}">
				<div class="form-group">
					<label for="id" class="col-sm-2 control-label">Article id</label>
					<div class="col-sm-5">
						<input th:field="${article.id}" type="text" id="id" name="id" class="form-control"
th:placeholder="#{article.id}" />
					</div>
				</div>
				<div class="form-group">
					<label for="name" class="col-sm-2 control-label">Article name</label>
					<div class="col-sm-5">
						<input th:field="${article.name}" type="text" id="name" name="name" class="form-control"
th:placeholder="#{article.name}" />
					</div>
				</div>
				<div class="form-group">
					<label for="price" class="col-sm-2 control-label">Article price</label>
					<div class="col-sm-5">
						<input th:field="${article.price}" type="text" id="price" name="price" class="form-control"
th:placeholder="#{article.price}" />
					</div>
				</div>
				<button type="submit" class="btn btn-primary">SEND</button>
			</form>
		</div>
	<nav><th:block th:insert="layout/default.html :: footernavbar"></th:block></nav>
	<div><th:block th:insert="layout/default.html :: scripts"></div>
</body>
</html>
```

fichier messages.properties

```
article.id= ID de l'article
article.name= Nom de l'article
article.price= Prix de l'article
article.error=Tous les champs doivent être complétés
basket.empty=Votre panier est vide
```

Spring - Application : showCatalog.html

```
<!DOCTYPE HTML>
<html lang="fr" xmlns:th="http://www.thymeleaf.org">
<head>
<th:block th:insert="layout/default.html :: headerfiles"></th:block>
<style>
table {
	font-family: arial, sans-serif;
	border-collapse: collapse;
	width: 100%;
}
td, th {
	border: 1px solid #dddddd;
	text-align: left;
	padding: 8px;
}
tr:nth-child(even) {
	background-color: #dddddd;
}
</style>
</head>
<body>
	<nav>
		<th:block th:insert="layout/default.html :: navbar"></th:block>
	</nav>
	<div class="container">
		<br> <br> <br>
		<h3>----- CATALOGUE -----</h3>
		<table>
			<tr>
				<th>ID</th>
				<th>NAME</th>
				<th>PRICE</th>
				<th>REMOVE</th>
				<th>BASKET +</th>
			</tr>
			<tr th:each="art : ${catalog}">
				<td><a th:href="@{'/showJson/'+${art.id}}" th:text="${art.id}"></a></td>
				<td id="artName" th:text="${art.name}"></td>
				<td th:text="${art.price}">59.9</td>
				<td><a th:href="@{'/delete/'+${art.id}}">Supprimer</a></td>
				<td><a th:href="@{'/addToBasket/'+${art.id}}">ajouter au panier</a></td>
			</tr>
		</table>
```

Spring - Application : showCatalog.html

```
<br>
            <div id="panier" th:if="${session.panier}" th:object="${session.panier}">
                  <h3>----- PANIER -----</h3>
                  <table>
                        <tr>
                              <th>ARTICLE</th>
                              <th>BASKET -</th>
                        </tr>
                        <tr th:each="art , rowStat: ${session.panier.articles}">
                              <td th:text="${art.name}"></td>
                              <td><a th:href="@{'/removeFromBasket/'+${rowStat.index}}">retirer du panier</a></td>
                        </tr>
                  </table>
                  <h5>Prix total</h5>
                  <span style="color: red;" th:text="${session.totalPrice}"></span>
                  <span style="color: red;"> € H.T.</span>
            </div>
            <div th:if="${session.emptyBasket}">
                  <br>
                  <h5 class="col-sm-3 alert alert-danger" th:text="#{basket.empty}"></h5>
            </div>
      </div>
      <nav>
            <th:block th:insert="layout/default.html :: footernavbar"></th:block>
      </nav>
      <div>
            <th:block th:insert="layout/default.html :: scripts">
      </div>
</body>
</html>
```

THE END