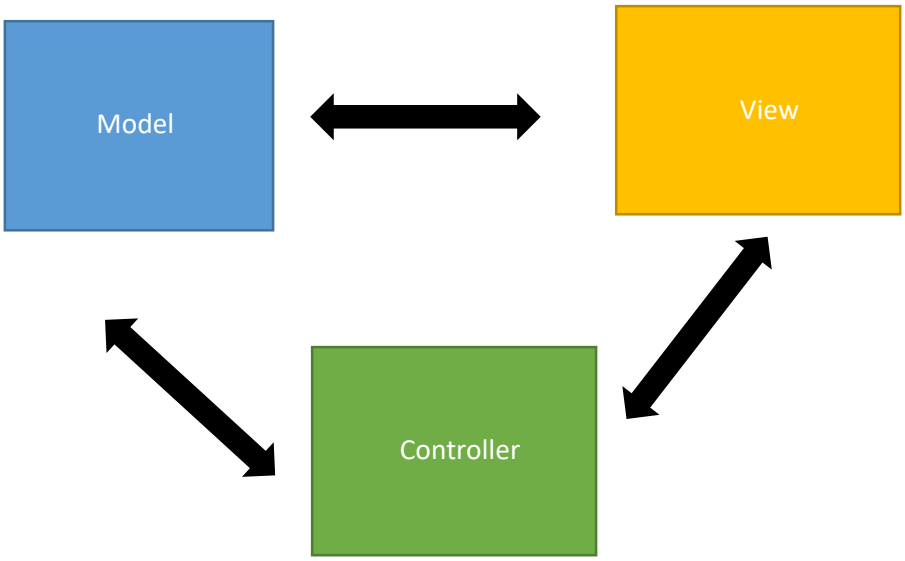
SpringMvc

Introduction

Comme son nom l'indique, Spring MVC est un modèle MVC : Model / View / Controller



Le Controller est le point d'entrée : il remplit le Model avec des données que la View affiche. Et réciproquement : s'il y a un formulaire dans la View, les données soumises sont injectées dans le Model puis traitées par le Controller.

Il s'agit d'un *modèle*, c.a.d d'une façon de gérer les applications web. Il existe d'autres modèles.

Il s'agit donc d'un design pattern (et non pas d'une obligation) pour coder des applications web, pour éviter de coder de façon anarchique.

- Le Controller est codé en Java, il appelle des services pour récupérer les données et remplir le Model.
- La View est codée en HTML avec des extensions permettant de faire des conditions, des boucles et des techniques pour afficher ce qu'il y a dans le Model.
- Le Model est fourni (ici par SpringMvc). C'est un espace mémoire, que l'on peut voir comme une Map< ?, ?>

Prenons un exemple d'une page qui affiche une liste d'Item.

- Le Controller appelle le service ItemService pour récupérer une List<Item>. Puis il met cette liste dans le Model
- La View récupère cette List<Item> dans le Model puis l'affiche (en HTML avec extensions)

A l'inverse imaginons qu'on crée un Item via un formulaire HTML

- Quand on appuie sur le bouton Submit, les données du formulaire sont stockées dans le Model et le Controller est appelé
- Le Controller récupère les données du Model et appelle le service pour créer l'Item en BDD.

Premiers pas

Une chose à savoir avant de commencer avec SpringBoot : contrairement à ce qu'on a vu auparavant, le projet web est un *jar* et non pas un *war*. En *développement*, SpringBoot se charge de gérer le serveur comme un grand, il suffit de lancer l'Application, comme d'habitude, ce qui simplifie le développement. Evidemment, en déploiement, il faudra générer un war, mais on verra ça plus tard.

Plutôt que bâtir un programme SpringMvc depuis zéro, on va partir de l'exemple de base fournit par SpringBoot : on aura comme cela l'architecture d'un programme Web SpringBoot et ce sera plus simple à comprendre pour commencer.

- Click droit puis *New > Import Spring getting Started Content*
- Sélectionner *Serving Web Content*
- Décocher *initial*
- Cliquer sur *Finish*
- Attendre un peu que Maven ait fini de bosser puis ouvrir le projet *gs-serving-web-content-complete*

Que voit-on ?

Dans *src/main/java*, on a le code Java :

- Une Application (qui lance le programme)
- Un Controller

Dans *src/main/resources,* on a le schéma classique d'une application MVC sous SpringBoot

- Un répertoire *static* qui contient les pages HTML statiques et tous les autres objets statiques (images, css, etc). Ici on a juste une page index.html qui est affichée par défaut.
- Un répertoire *templates* qui contient toutes les pages HTML « dynamiques », c.a.d en HTML + extensions. La library des extensions s'appelle *ThymeLeaf*, et on la verra plus en détails après. Le répertoire *templates* contient donc les View.

Avant de nous intéresser au code, lançons déjà le programme :

- Click-droit sur Application
- *Run as > Spring Boot App*
- Dans le browser, taper *localhost :8080*
- Cliquer sur le lien
- On arrive sur une autre page (*http://localhost:8080/greeting*) qui affiche *Hello World*
- Si dans la barre d'url on tape http://localhost:8080/greeting?name=toto, la page affiche *Hello toto*

Regardons déjà la View (*greeting.html*) :

```
Pour dire qu'on est en HTML 5
<!DOCTYPE HTML>
Pour dire qu'on utilise thymeleaf
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Getting Started: Serving Web Content</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
</head>
<body>
```

```
Ici on a une instruction thymeleaf qui dit d'afficher le truc qui s'appelle name dans le Model.
Pour être exact on affiche une concaténation de Hello et de ${name}, soit le truc qui s'appelle name
dans le Model.
<p th:text="'Hello, ' + ${name} + '!'" />
</body>
</html>
```

Voyons maintenant le Controller

```
Indique que c'est un Controller
@Controller
public class GreetingController {

    Là, c'est la partie intéressante : on indique que si le serveur est appelé avec l'url /greeting,
alors la fonction greeting (celle-ci) est appelée (seulement si c'est un GET, dans ce cas)
    @GetMapping("/greeting")
    On indique que la fonction greeting peut recevoir une @RequestParam
    public String greeting(@RequestParam(name="name", required=false, defaultValue="World") String
name, Model model) {
        Mise à jour du Model avec un attribut de nom name et de valeur name
        model.addAttribute("name", name);
        On retourne le nom de la template (la View) à afficher en sortie
        return "greeting";
    }

}
```

Avant d'aller plus loin allons voir ce qu'il y a dans index.html. La partie intéressante est ceci :
`<p>Get your greeting <a href="/greeting">here</a></p>`

On voit qu'on a un lien sur */greeting* : quand on clique dessus, le serveur est donc appelé avec l'url */greeting*, et donc la fonction greeting() du Controller va être appelée.

On sait que si on tape à la fin de l'url ?name=toto, on affichera Hello toto. Dans une url, tout ce qu'il y a après le ? sont des paramètres de requête. D'où le paramètre `(@RequestParam(name="name", required=false, defaultValue="World")` qui indique qu'on passe à la fonction un paramètre de request de nom name et si elle n'est pas présente, le défaut est World.

Changeons le nom du paramètre dans le Controller de *name* en *nom*. Puis tapons dans l'url : http://localhost:8080/greeting?nom=toto

Voyons la suite de la fonction du Controller : on ajoute donc dans le Model un attribut de nom *name* avec comme valeur ce qu'il y a dans la variable *name*.

Comme tout cela n'est pas très clair, changeons dans le Controller, le nom de l'attribut de *name* en *affichage*, et de même dans *greeting.html*.

Le Controller

Avant d'aller plus loin, il faut voir plusieurs choses.

Les urls

Il est de bon ton, actuellement, d'utiliser des url REST ou à la façon de.

- GET /specie : renvoie la liste des Specie
- GET /specie/2 : renvoie le Specie d'id 2
- POST /specie/create : crée un Specie
- POST /specie/2/update : update un Specie
- GET /specie/2/delete : détruit le Specie d'id 2

Tout cela pour dire qu'on évite d'utiliser des requestParam avec des url du genre GET /specie ?action=delete&id=2.

Cela change au niveau du Controller. On n'utilise donc plus des @RequestParam mais des @PathParam, comme par exemple :

```
@GetMapping("/specie/{id}/delete")
public String delete(@PathVariable("id") long id) {
```

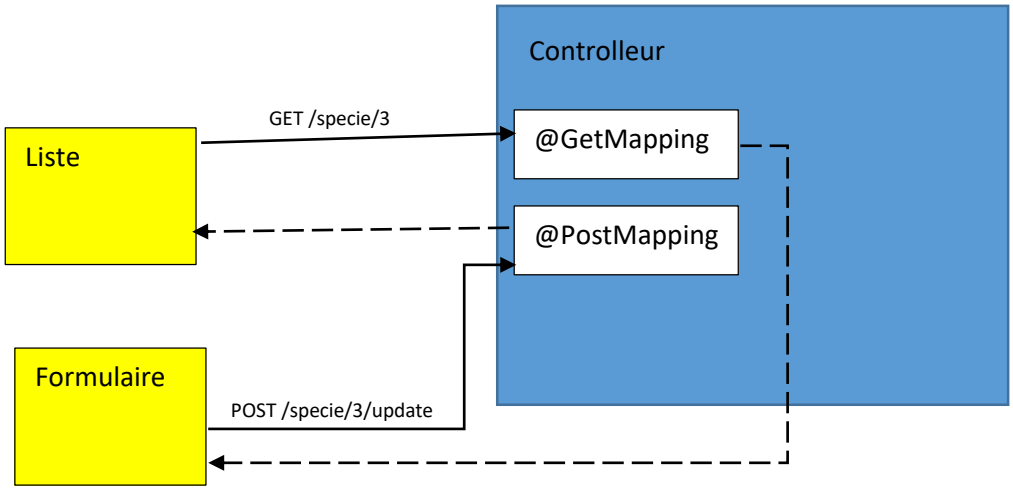
POST et GET

Il n'est plus nécessaire de passer par des pages d'indirection cachées. La logique est plutôt la suivante :

- Pour afficher une page, on fait un GET
- Pour traiter les données de cette page, on fait un POST

Imaginons qu'on vienne de la page de la liste des Specie et qu'on veuille updater un Specie en particulier :

- On fait GET /specie/3 : on affiche un formulaire avec les champs pré-remplis avec les valeurs du Specie d'id 3. Donc le Controller a une fonction associée via @GetMapping ("/specie/{id}") qui récupère le Specie d'id 3 et le met dans le Model et enfin affiche la page de formulaire.
- Quand on clique sur Submit, on fait POST /specie/3/update : le Controller a une fonction associée via @PostMapping ("/specie/{id}/update") qui récupère les données du Model et écrit dans la BDD puis réaffiche la page de liste.



Les retours

On a vu que les méthodes du Controller renvoyaient le nom de la template à afficher en sortie. Elles peuvent aussi renvoyer une redirection : `return "redirect:/specie"` qui en pratique appelle la méthode du Controller qui est mappée sur cette url.

Le Model

Comme on l'a vu, le Model est une sorte de Map< ?, ?>. D'ailleurs, on peut utiliser le Model comme une Map< ?, ?>, mais généralement on utilise la fonction :

- model.addAtribute(nom_clé,valeur) : met une valeur dans le Model et l'associe à une clé. La View récupérera donc cette valeur via la clé.

On peut récupérer le Model sous forme de Map< ?, ?> via model.asMap() ce qui permet d'effectuer toutes les opérations possibles sur une Map< ?, ?> (mais c'est rarement utile).

ThymeLeaf

Voici le lien vers la documentation officielle de thymeleaf, car ne sera présenté ici que le minimum : https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html

A noter que SpringMvc fournit aussi quelques extensions à Thymeleaf qu'on ne verra pas ici.

Utiliser un élément du modèle

${nom_de_l_attribut}

On peut accéder aux attributs d'un objet s'il est dans le Model. Soit un objet Animal dans l'attribut bestiole. ${bestiole.specie.latinName} accède au champ latinName de specie de bestiole (qui est de Class Animal).

Afficher un élément du modèle

On utilise th:text

```
<span th:text="${bestiole.specie.latinName}"></span>
```

```
<span th:text="'Le nom de la bestiole est : '+${bestiole.specie.latinName}"></span>
```

Condition

th:if

```
<div th:if="${value}"><span>Je m'affiche si value est true</span></div>
```

```
<div th:if="${firstName == 'Paul'}"><span>Je m'affiche si le prénom est Paul</span></div>
```

```
<div th:if="${person.age < 20}"><span>Je m'affiche si l'age est inférieur à 20</span></div>
```

Boucle

th:each

```
<tr th:each="specie : ${species}">
…
</tr>
```

Lien

Pour les href et les src des img, utiliser th:href= "@{url}" et th:src.

```
th:href="@{/resources/css/main.css}"
```

```
th:src="@{/resources/img/chat.jpg}"
```

Il s'agit des url réelles et non pas des chemins.

A noter un truc important : si on a une architecture du genre :

+src/main/resources
++static
+++resources

++++img
++++css
++++js
++templates

Alors resources (le 2eme) est en fait au même niveau que templates au niveau des url ce qui explique le th:src et th:href plus haut.

Si on veut appeler une fonction javascript :

```
<a th:href="'javascript:editFunc(' + ${specie.id} +');'">…</a>
```

Formulaire

Au niveau d'un formulaire, on peut travailler directement au niveau d'un objet qui sera automatiquement lu et mis à jour via le Model.

Supposons qu'on ait un objet Person mis dans le Model avec l'attribut *personne*

```
<form th:object="${personne}" th:action="@{/specie/create}" method="post">
    <input type="text" th:field="*{firstName}"/><br/>
    <input type="text" th:field="*{lastName}"/><br/>
    <input type="submit" class="button" th:value="Save"/>
</form>
```

A l'affichage, la <form> va afficher l'objet Person.

*{field} correspond au champ field de l'objet Person (ici *{firstName} est équivalent à ${personne.firstName}).

th:field est un shortcut qui en pratique va créer le id, le name et la value du champ associé.
Soit th:field="*{firstName}". Il va générer

- Id égal à 'firstName'
- Name égal à 'firstName'
- Value égal à la valeur de personne.firstName

Ne pas oublier de mettre un objet même vide. Par exemple en création, quand les champs sont vides, il faut que ${personne} contienne une Person vide (et pas rien).

Du côté du Controller, la méthode est quelque chose comme ça :

```
    @PostMapping("specie/create")
    public String doCreate(Specie sp, Model model) {
```

Vu que la <form> travaille directement avec l'objet, la méthode (POST) du Controller peut directement récupérer cet objet rempli avec les valeurs de la <form>.

ThymeLeaf étendu

Variables prédéfinies

Thymeleaf propose des objets prédéfinis de la forme #var.
Par exemple ${#locale} correspond à la Locale courante.

Voir la doc pour la liste de ces variables

Variables

On peut définir des variables avec th:with.
Exemple :

```
<div th:with="lang='La langue est : '+${#locale.language}">
<span th:text="${lang}"></span>
</div>
```

On voit que ces variables ont un scope. Dans le cas ci-dessus, elle n'est valable que dans la balise

Internationalisation

D'une manière générale, il ne faut **jamais** mettre en dur des chaines dans des pages HTML. Cela permet d'avoir un site web multi-langues.

On commence par créer un répertoire *messages* dans *src/main/resources*. Dedans on crée des fichiers properties contenant toutes les chaines qui apparaissent dans le site web.
Le nom des fichiers est de la forme *messages_codepays.properties*.

messages_fr.properties :
bonjour=Bonjour
au_revoir=Au revoir

messages_en.properties :
bonjour=Hello
au_revoir=Good bye

messages_es.properties
bonjour=Buenos dias
au_revoir=Hasta luego

Et dans la page HTML on a :

Noter le # qui signifie qu'il s'agit d'un texte externalisé.

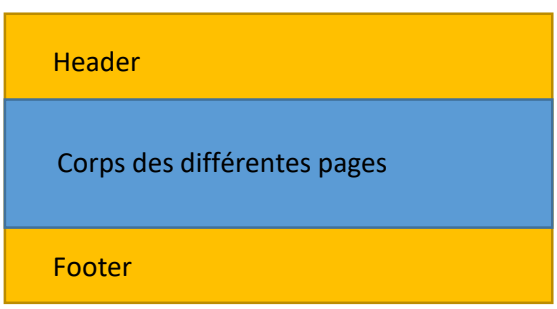
La langue du site web est déterminée parce qu'on appelle la Locale qui est propre au système d'exploitation, à Java, à des variables d'environnement, etc. Il est possible de changer la Locale via le site, en Implémentant un WebMvcConfigurer ; mais on ne verra pas cela pour l'instant.

A noter qu'il faut aussi mettre un fichier messages.properties (sans extension) qui est le fichier de message par défaut ou celui qui est utilisé si on ne reconnait pas la Locale.

Fragments

Là, ça devient compliqué. Il y a deux types d'utilisation des fragments :

1) On veut que toutes les pages d'un site aient le même header (un bandeau, un menu, etc) et le même footer (des indications diverses).



Voyons déjà le Fragment (ce qui ne bouge pas, en orange sur le dessin) :

```
Ici on définit tout le html comme fragment, ce qui est logique, puisque c'est la page elle-même. Il est donc défini au niveau de <html>. On donne le nom du fragment (ici layout) et les paramètres (ici template, le code qui va être mis au milieu).
<html th:fragment="layout (template)">
Logique, on définit le <head> à ce niveau là, puisque c'est toute la page.
<head>
…
</head>
  <body>
        …….
        ICI LE HEADER
        ……
        On inclut ici la partie qui bouge, donc template qui a été passé en paramètre.
        <th:block th:include="${template}"/>
        …….
        ICI LE FOOTER
        ……
  </body>
</html>
```

Voyons une page qui va être incluse (en bleu sur le dessin) :

```
On déclare l'utilisation du fragment au niveau du <html> puisque toute la page va être remplacée par le fragment. Il faut indiquer le chemin du fragment (fragments/layout), puis le nom du fragment (layout), car il peut en théorie y avoir plusieurs Fragments par fichier de fragment, et enfin ce qui doit être inséré dans le fragment : on utilise ici la variable ~{::body} qui indique qu'on insère toute la balise <body> de la page courante.
<html xmlns:th="http://www.thymeleaf.org" th:replace="~{fragments/layout :: layout (~{::body})}">
On remarque qu'il n'y a pas de <head> puisque c'est dans le Fragment
  <body>
  ….. Le corps de la page …..
  </body>
</html>
```

2) Un fragment peut aussi servir comme une sorte d'objet graphique réutilisable dans plusieurs pages.

   Disons qu'on crée un fragment composé d'un label et d'un champ d'édition.

Dans la page qui utilise ce fragment, on a ceci :

```
<input th:replace="~{fragments/edit_a_moi :: mon_edit (#{latinName},'latinName')}" />
```

Donc le <input> va être remplacé par le fragment (on peut utiliser autre chose qu'un <input>, mais comme c'est dans une <form>, autant utiliser un <input>).
Les paramètres sont un message (le titre) et le nom du champ à utiliser. On ne peut pas utiliser *{latinName} (du fait de l'implémentation de thymeleaf), et on passe donc le nom du champ comme une chaine.

Dans le fragment, on a :

```
<html>
<body>
  <form>
    <th:block th:fragment="mon_edit(label,name)">
    <label th:text="${label}"></label> <input type="text" th:field="*{__${name}__}"/>
    </th:block>
    </form>
    </body>
</html>
```

Seule la partie surlignée est vraiment utile. On met le <body> et le <form> pour que ce soit une page HTML cohérente.
On voit que pour afficher le label, on met *${label}*, ce qui est cohérent vu que label est passé en paramètre.
Pour le th:field, c'est plus compliqué. On sait qu'il faut qu'on ait un truc du genre *{}, mais comme on a passé le nom du champ sous forme de chaine, on ne peut pas directement écrire **{${name}}* ; il faut passer par le précompilateur pour que thymeleaf retrouve ses petits, donc **{__${name}__}* (voir la doc pour plus de détails).

Validator

Le validateur est utilisé typiquement lorsque des valeurs remontent d'un formulaire et qu'il faut tester la validité de ces valeurs.

Le Validator se positionne donc entre la View et le Controler.

Actuellement, on utilise de moins en moins de Validator, les contrôles se faisant maintenant en JS dans la View. Toutefois, il peut arriver que dans des cas un peu complexes, il faille utiliser un Validator.

```
Dans le controller, On initialise le Binder avec un Validator qu'on va créer plus loin. Le Binder est le truc qui fait le lien entre le Model et les données remontant du Browser.
@InitBinder
    public void initSpecieBinder(WebDataBinder dataBinder) {
        dataBinder.setValidator(new SpecieValidator());
}

Dans un Mapping du controller, on signale qu'une donnée est à valider, ici le premier paramètre. On indique que ce paramètre est à valider avec l'annotation @Valid.
@PostMapping("specie/create")
    public String doCreate(@Valid Specie sp, BindingResult result,Model model) {
        Ici on teste si le retour de la validation a des erreurs ou pas et on retourne en mettant à jour éventuellement le Model.
        if (result.hasErrors()) {
                model.addAttribute("specie", sp);
                return "create";
        }

…

On crée donc un Validator.
public class SpecieValidator implements Validator {

        @Override
        Là on dit que seules les Class de type Specie (ou ses dérivées) sont acceptées
        public boolean supports(Class<?> clazz) {
                return Specie.class.isAssignableFrom(clazz);
        }

        Là, on valide effectivement les valeurs (on teste que latinName et commonName ne sont pas nuls)
        @Override
```

```
        public void validate(Object target, Errors errors) {
                Specie sp = (Specie)target;
                if (sp.getLatinName() == null || sp.getLatinName().isEmpty()) {
                        En cas d'erreur, on rejette. Remarquons qu'on ne sort pas tout de suite mais
                        qu'on teste toutes les valeurs.
                        errors.rejectValue("latinName", "required","required");
                }
                if (sp.getCommonName() == null || sp.getCommonName().isEmpty()) {
                        errors.rejectValue("commonName", "required","required");
                }
        }

}
```

Et dans le code HTML, on a (par exemple):

```
On teste qu'il y a des erreurs et on affiche le texte de l'erreur (ici "required") pour le champ
considéré.
<span th:if="${#fields.hasErrors(latinName)}" th:errors="*{latinName}" class="errorText">Error</span>
```

Attention : ici, la variable prédéfinie *#fields* est propre aux extensions Spring pour Thymeleaf.