Le modèle M.V.C. de Spring 1/2

1 Introduction

Le framework **Spring boot** a pour objectif d'accélerer et de simplifier la mise en place d'une application basée sur Spring.

Cet objectif est atteint par l'intégration automatique de nombreux composants et un système d'auto-configuration qui couvre la majorité des cas.

1.1 Mise en place d'un projet

Préparez un environement pour tester une application WEB basée sur Spring-boot :

- Téléchargez le projet Maven [1] déjà préparé.
- Décompressez cette archive.
- Importez, dans Eclipse, un projet Maven et choisissez le répertoire précédent.
- Exécutez ce projet (**Run as .. Java application** classe `mybootapp.Starter`).
- Testez le bon fonctionnement à l'adresse `http://localhost:8081`

Explication : Spring boot lance une instance embarquée de Tomcat pour déployer votre application WEB. Elle est donc directement accessible.

À faire :

- Stopper l'exécution.
- Faites un **Run as ... Maven build... goal : package** dans Eclipse.
- Faites la même chose en ligne de commande :

Build du projet

```
cd repertoire_de_votre_projet
mvn package
```

- Ces deux opérations ont généré une fichier `.war` dans le répertoire `target`.
- Lancez votre application en ligne de commande :

Exécution du projet

```
cd repertoire_de_votre_projet
java -jar target/nom_de_votre_fichier.war
```

- **Moralité** : il est très simple de compiler, déplacer et exécuter une application WEB avec ce type d'outil.

1.2 Explorer ce projet

Les projets **Maven** ont une structure particulière :

- `pom.xml` : configuration de Maven

1. jeeWebApp.zip

- `src/main/java` : le code source Java
- `src/main/resources` : les ressources (fichiers XML, images, propriétés, etc.)
- `src/main/resources/static` : les ressources statiques de votre application WEB (CSS, images, autres)
- `src/main/resources/application.properties` : le fichier de paramétrage de Spring boot
- `src/test/java` : le code source Java de test
- `src/test/resources` : les ressources pour le test
- `src/main/webapp` : les fichiers de l'application WEB (pages JSP par exemple)
- `src/main/webapp/WEB-INF` : le répertoire `WEB-INF`
- `src/main/webapp/WEB-INF/jsp` : les pages JSP cachées

Travail à faire :

- Faites fonctionner l'application WEB,
- Explorez le contrôleur `MyControler` et la page `src/main/webapp/WEB-INF/jsp/index.jsp`. Changez le message dans le fichier `src/main/resources/application.properties` et vérifiez sa prise en compte.
- Explorez le contrôleur `CourseControler` et la page `src/main/webapp/WEB-INF/jsp/course.jsp`.
- **Remarque** : Les pages JSP sont cachées dans le répertoire `WEB-INF/jsp`. Elles ne sont donc pas accessibles via une URL construite par le client. Ce dernier doit donc passer obligatoirement par un contrôleur pour envoyer une requête.
- Explorez l'interface DAO : `CourseRepository`.
- Explorez la classe de démarrage : `Starter`. **Explication** : La méthode `main` initialise l'application Spring. Ce dernier va créer une servlet générique (`DispatcherServlet`) qui va récupérer toutes les requêtes et les aiguiller vers le bon contrôleur.
- Préparez un onglet vers la documentation [a].
- Préparez un onglet vers la Javadoc [b].

a. https ://docs.spring.io/spring-framework/reference/index.html
b. https ://docs.spring.io/spring/docs/6.0.14/javadoc-api/index.html

2 Mise en place de Spring MVC

2.1 Un premier contrôleur

== Cette classe Spring va créer un `bean` et lui donner un nom (`/hello`). C'est notre premier contrôleur.

```java
package mybootapp.web;

import java.io.IOException;

import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.stereotype.Service;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;

@Service("/hello")
public class HelloController implements Controller {

    protected final Log logger = LogFactory.getLog(getClass());

    public ModelAndView handleRequest(HttpServletRequest request,
            HttpServletResponse response)
            throws ServletException, IOException {
        logger.info("Returning␣hello␣view");
        return new ModelAndView("hello");
    }

}
```

== À cette étape, vous devez trouver dans les traces du serveur la création de ce contrôleur (ajoutez une méthode annotée `@PostConstruct`).

== Essayez d'utiliser le contrôleur `hello`.

== Visiblement ce contrôleur se termine en donnant la main à une page JSP (`src/main/webapp/WEB-INF/jsp/hello.jsp`) destinée à fabriquer la réponse à envoyer au client. Créons cette page :

Page `src/main/webapp/WEB-INF/jsp/hello.jsp`

```jsp
<html>
  <head><title>Hello :: Spring Application</title></head>
  <body>
    <h1>Hello - Spring Application</h1>
  </body>
</html>
```

== Le passage entre le nom de la page `hello` et sa position (`src/main/webapp/WEB-INF/jsp/hello.jsp`) est résolu par les paramètres `spring.mvc.view.prefix` et `spring.mvc.view.suffix` qui se trouvent dans le fichier de configuration `application.properties`.

Travail à faire : Étudiez la classe `ModelAndView` [a]. Cette classe est le coeur du modèle MVC de Spring : Elle permet de séparer d'une part, les contrôleurs qui travaillent sur la requête et d'autres part les vues (pages JSP) qui se chargent du résultat. Entre les deux, les instances de `ModelAndView` transportent à la fois le nom de la vue et les données qui seront affichés par cette vue (c'est-à-dire le **modèle**).

a. https ://docs.spring.io/spring/docs/6.0.14/javadoc-api/org/springframework/web/servlet/ModelAndView.html

2.2 Améliorer les vues

== Nous avons déjà prévu dans le fichier `pom.xml` les dépendances pour utiliser la JSTL.

== C'est la même chose pour bootstrap via les fichiers `webjars`. Nous avons également prévu un fichier `WEB-INF/jsp/header.jsp` pour les déclarations d'ouverture d'une page HTML et `WEB-INF/jsp/footer.jsp` pour les déclarations de fermeture. Explorez ces fichiers. Nous pouvons donc simplifier notre page JSP (nous avons ajouté un `div` pour créer un conteneur bootstrap ainsi que l'affichage d'un paramètre `now`) :

Page `src/main/webapp/WEB-INF/jsp/hello.jsp`

```
<%@ include file="/WEB-INF/jsp/header.jsp" %>

<h1>Hello - Spring Application</h1>
<p>Greetings, it is now <c:out value="${now}" default="None" /></p>

<%@ include file="/WEB-INF/jsp/footer.jsp" %>
```

== Bien entendu, nous devons modifier le contrôleur en conséquence : le contrôleur fabrique maintenant une donnée et transmet cette donnée à la page JSP qui se charge de l'afficher (la date).

```
...
    String now = (new Date()).toString();
    return new ModelAndView("hello", "now", now);
...
```

Travail à faire : Préparez une nouvelle donnée (par exemple un message), passez cette donnée à la page `hello.jsp` et assurez sa présentation.

3 Utiliser les annotations

== Nous pouvons maintenant définir un nouveau contrôleur :

```java
package mybootapp.web;

import java.util.Date;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;

@Controller()
@RequestMapping("/tests")
public class HelloAnnoController {

    protected final Log logger = LogFactory.getLog(getClass());

    @GetMapping("/welcome")
    public ModelAndView sayHello() {
        String now = (new Date()).toString();
        logger.info("Running␣" + this);
        return new ModelAndView("hello", "now", now);
    }

}
```

Travail à faire :

- Vérifiez dans les traces du serveur que ces contrôleurs sont bien détectés par Spring.
- Testez ce contrôleur avec une URL du type

  `http://localhost:8081/tests/welcome`

- Lisez la documention de l'annotation `@Controller`.
- Un contrôleur est maintenant une méthode qui
  - ▷ renvoie une instance de `ModelAndView` ou le nom d'une vue (`String`),
  - ▷ accepte en argument une instance de `HttpServletRequest` et/ou `HttpServletRequest` et/ou `HttpSession` et bien d'autres choses.
- Créez un nouveau contrôleur (`/tests/counter`) qui va stocker un compteur en session (ajoutez un paramètre `HttpSession`), le faire évoluer et assurer son affichage. Faites en sorte que ce contrôleur traite également un argument de la requête HTTP.

3.1 Déclarer les paramètres

Nous pouvons également utiliser l'annotation `@RequestParam` pour récupérer, sous la forme d'un paramètre de la méthode, les paramètres de la requête HTTP. En voici un exemple :

```java
@GetMapping("/plus10")
public ModelAndView plus10(
        @RequestParam(value = "num", defaultValue = "100") Integer value) {
    logger.info("Running␣plus10␣controler␣with␣param␣=␣" + value);
    return new ModelAndView("hello", "now", value + 10);
}
```

Travail à faire :

- Testez ce contrôleur en lui fournissant le paramètre attendu. Testez également les cas d'erreur (paramètre absent ou incorrect).
- Ajoutez un nouveau paramètre de type `Date` et utilisez l'annotation `@DateTimeFormat` pour récupérer ce paramètre.

3.2 Utiliser de belles adresses

Il est maintenant habituel de placer des paramètres à l'intérieur des adresses WEB. cela permet d'avoir des URL simples, faciles à construire et faciles à mémoriser. En voila un exemple :

```java
@GetMapping("/voir/{param}")
public ModelAndView voir(@PathVariable("param") Integer param) {
    logger.info("Running␣param␣controler␣with␣param=" + param);
    return new ModelAndView("hello", "now", param);
}
```

Travail à faire :

- Testez ce contrôleur.
- Modifiez ce contrôleur pour avoir plusieurs paramètres dans la même adresse.
- Utilisez le mécanisme des expressions régulières pour traiter une adresse composée (inspirez de la documention sur l'annotation `@PathVariable` [a]).

a. https ://docs.spring.io/spring-framework/reference/web.html#mvc-ann-requestmapping-uri-templates

3.3 Variables matrix

Nous pouvons également traiter des URL de la forme `/une/action;a=1;b=2`, c'est ce que nous appelerons les **variables matrix**.

Commencez par enrichir la classe de configuration (`Starter`) en ajoutant la méthode ci-dessous (`configurePathMatch` est définie dans l'interface `WebMvcConfigurer` [2]) :

Ajout dans `Starter.java`

```java
@Override
// Pour activer les variables matrix
public void configurePathMatch(PathMatchConfigurer configurer) {
    var urlPathHelper = new UrlPathHelper();
    // Nous gardons le contenu après le point virgule
    urlPathHelper.setRemoveSemicolonContent(false);
    configurer.setUrlPathHelper(urlPathHelper);
}
```

2. https ://docs.spring.io/spring/docs/6.0.14/javadoc-api/org/springframework/web/servlet/config/annotation/WebMvcConfigurer.html

Ajoutez ensuite un nouveau contrôleur pour tester ce mécanisme :

```
@GetMapping("/matrix/{param}")
@ResponseBody
public String testMatrix(//
        @PathVariable("param") String param, //
        @MatrixVariable(name = "a", defaultValue = "A") String a, //
        @MatrixVariable(name = "b", defaultValue = "1") Integer b//
) {
    return String.format("param=%s, a=%s, b=%d", param, a, b);
}
```

Note : Nous profitons de cet exemple pour illustrer l'annotation `@ResponseBody` qui permet d'expliquer que la méthode renvoie la réponse et non pas le nom d'une vue.

Travail à faire :

- Testez ce contrôleur avec l'URL `/tests/matrix/hello;a=AA`
- Essayez plusieurs variantes.
- Essayer d'utiliser un seul paramètre matrix (sans nom) de type `Map<String,String>` pour récupérer toutes les affectations.

4 Le traitement des formulaires

Pour introduire la traitement des formulaires nous allons procéder en trois étapes :

- définition d'un modèle et d'une couche métier,
- création du formulaire,
- validation des données,

4.1 Définir un produit

Créez la classe JPA pour représenter un produit :

```java
package mybootapp.model;

import jakarta.persistence.Basic;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.Id;

import lombok.Data;

@Entity
@Data
public class Product {

    @Id
    @GeneratedValue
    private Long number;

    @Basic
    private String name;

    @Basic
    private Double price;

    @Basic
    private String description;

    @Basic
    private String type;

}
```

4.2 Définir la couche DAO

Créez ensuite l'interface DAO basée sur **Spring-data** :

```java
package mybootapp.repo;

import org.springframework.data.jpa.repository.JpaRepository;

import mybootapp.model.Product;

public interface ProductRepository extends JpaRepository<Product, Long> {

}
```

4.3 Lister les produits

Nous pouvons maintenant mettre en place un contrôleur qui va gérer toutes les actions sur les produits (listage, création, modification et suppression). Commençons par lister les produits disponibles :

```java
package mybootapp.web;

import java.util.Collection;

import jakarta.annotation.PostConstruct;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;

import mybootapp.model.Product;
import mybootapp.repo.ProductRepository;

@Controller
@RequestMapping("/product")
public class ProductController {

    @Autowired
    ProductRepository repo;

    protected final Log logger = LogFactory.getLog(getClass());

    @PostConstruct
    public void init() {
        Product p1 = new Product();
        p1.setName("Car");
        p1.setPrice(2000.0);
        p1.setDescription("Small␣car");
        p1.setType("x");
        Product p2 = new Product();
        p2.setName("Gift");
        p2.setPrice(100.0);
        p2.setDescription("Big␣gift");
        p2.setType("x");
        repo.save(p1);
        repo.save(p2);
    }

    @GetMapping("/list")
    public ModelAndView listProducts() {
        logger.info("List␣of␣products");
        Collection<Product> products = repo.findAll();
        return new ModelAndView("productsList", "products", products);
    }

}
```

Ce contrôleur est accompagné de la vue `productsList.jsp` :

```
<%@ include file="/WEB-INF/jsp/header.jsp"%>

<h1>Products</h1>

<table class="table table-hover">
    <c:forEach items="${products}" var="prod">
        <c:url var="edit" value="/product/edit">
            <c:param name="id" value="${prod.number}" />
        </c:url>
        <tr>
            <td><a href="${edit}"><c:out value="${prod.name}" /></a></td>
            <td><i>$<c:out value="${prod.price}" /></i></td>
        </tr>
    </c:forEach>
</table>

<c:url var="create" value="/product/edit" />
<p>
    <a class="btn btn-info" href="${create}">Create new product</a>
</p>

<%@ include file="/WEB-INF/jsp/footer.jsp"%>
```

Cette vue va construire la liste des produits, avec pour chacun une possibilité d'édition. Bien entendu, pour l'instant, la phase d'édition ne fonctionne pas.

Note : Dans cet exemple, le contrôleur ne fait rien d'autre que de construire des données (la liste des produits) pour les envoyer à la vue. La création des données peut être découplée des contrôleurs et placée dans des méthodes annotées par `@ModelAttribute`. Ces méthodes sont systématiquement exécutées avant les contrôleurs pour remplir le modèle.

Dans notre exemple, la création de la liste des produits (nommée `products`) peut se faire par la méthode :

```
@ModelAttribute("products")
Collection<Product> products() {
    logger.info("Making␣list␣of␣products");
    return repo.findAll();
}
```

Le contrôleur devient :

```
@GetMapping("/list")
public String listProducts() {
    logger.info("List␣of␣products");
    return "productsList";
}
```

La vue va simplement puiser dans le modèle qui est rempli par la méthode `products`.

4.4 Éditer un produit

Définissons maintenant le contrôleur d'accès au formulaire d'édition :

```
@GetMapping("/edit")
public String editProduct(@ModelAttribute Product p) {
    return "productForm";
}
```

accompagné du formulaire `productForm.jsp` :

```
<%@ include file="/WEB-INF/jsp/header.jsp"%>

<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>

<h1>Edit Product</h1>

<div class="card bg-light">
    <div class="card-body">
        <form:form method="POST" modelAttribute="product">

            <form:errors path="*" cssClass="alert alert-danger" element="div" />

            <div class="form-group my-1">
                <label for="name">Name:</label>
                <form:input class="form-control" path="name" />
                <form:errors path="name" cssClass="alert alert-warning"
                    element="div" />
            </div>
            <div class="form-group my-1">
                <label for="description">Description:</label>
                <form:textarea class="form-control" path="description" rows="4" />
                <form:errors path="description" cssClass="alert alert-warning"
                    element="div" />
            </div>
            <div class="form-group my-1">
                <label for="price">Price:</label>
                <form:input path="price" class="form-control" />
                <form:errors path="price" cssClass="alert alert-warning"
                    element="div" />
            </div>
            <div class="form-group my-1">
                <button type="submit" class="btn btn-info">Submit</button>
            </div>
        </form:form>
    </div>
</div>

<%@ include file="/WEB-INF/jsp/footer.jsp"%>
```

Cette vue utilise les balises personnalisées de Spring pour gérer facilement la récupération des données du modèle (attribut `modelAttribute` de la balise `form`) et la mise en place des champs (balises `form:input`, `form:select`, etc...). Vous trouverez plus d'information sur ces balises dans cette documentation [3].

Pour l'instant, ce formulaire ne permet pas d'éditer des produits déjà existants. Pour ce faire, nous allons ajouter une méthode annotée `@ModelAttribute` qui va préparer l'instance du produit à éditer en fonction du paramètre de la requête HTTP :

3. https://docs.spring.io/spring-framework/reference/web/webmvc-view/mvc-jsp.html#mvc-view-jsp-formtaglib

```java
@ModelAttribute
public Product newProduct(
    @RequestParam(value = "id", required = false) Long productNumber)
{
    if (productNumber != null) {
        logger.info("find product " + productNumber);
        var p = repo.findById(productNumber);
        return p.get();
    }
    Product p = new Product();
    p.setNumber(null);
    p.setName("");
    p.setPrice(0.0);
    p.setDescription("");
    logger.info("new product = " + p);
    return p;
}
```

En clair : si la requête `/edit` est accompagnée d'un numéro de produit (paramètre `id` optionnel), le produit sera chargé à partir du manager. Dans le cas contraire, un nouveau produit sera renvoyé. Testez ce fonctionnement.

Il nous reste maintenant à mettre en place le contrôleur de soumission du formulaire :

```java
@PostMapping("/edit")
public String saveProduct(@ModelAttribute Product p, BindingResult result) {
    if (result.hasErrors()) {
        return "productForm";
    }
    repo.save(p);
    return "productsList";
}
```

A cette étape, la seule erreur possible provient d'une erreur de conversion sur le prix. Essayez de donner un prix incorrect afin de tester ce fonctionnement.

Note : À ce stade, la création d'un nouveau produit (après la soumission) se termine sur l'affichage de la liste des produits (dernière ligne du contrôleur ci-dessus). Ce comportement pose un problème : Si le client tente un rechargement de la page, cela va provoquer une nouvelle soumission et la création d'un nouveau produit ! Pour régler ce problème, nous allons renvoyer non pas sur la vue `productsList`, mais sur l'action permettant d'avoir la liste des produits :

```java
@PostMapping("/edit")
public String saveProduct(@ModelAttribute Product p, BindingResult result) {
    if (result.hasErrors()) {
        return "productForm";
    }
    repo.save(p);
    return "redirect:list";
}
```

4.5 Injecter des données

Pour construire un formulaire complexe, nous avons souvent besoin d'utiliser des données annexes (liste de références, nom d'utilisateur, etc.). Pour ce faire, nous allons de nouveau utiliser l'annotation `@ModelAttribute`. Mettez en place la méthode suivante :

```java
@ModelAttribute("productTypes")
public Map<String, String> productTypes() {
    Map<String, String> types = new LinkedHashMap<>();
    types.put("type1", "Type␣1");
    types.put("type2", "Type␣2");
    types.put("type3", "Type␣3");
    types.put("type4", "Type␣4");
    types.put("type5", "Type␣5");
    return types;
}
```

Elle fabrique et injecte dans le modèle une table de correspondance qui va nous être utile pour ajouter le champ de typage dans le formulaire. Modifiez notre formulaire en ajoutant :

```jsp
<div class="form-group my-1">
    <label for="type">Type:</label>
    <form:select path="type" multiple="false" class="form-control">
        <form:option value="" label="--- Select ---" />
        <form:options items="${productTypes}" />
    </form:select>
    <form:errors path="type" cssClass="alert alert-warning"
        element="div" />
</div>
```

Nous pouvons maintenant associer un type à chaque produit.

4.6 Valider les données

Il manque maintenant la phase de validation des données du formulaire. Pour ce faire, nous allons développer une classe de validation adaptée au produit :

```java
package mybootapp.web;

import org.springframework.stereotype.Service;
import org.springframework.validation.Errors;
import org.springframework.validation.ValidationUtils;
import org.springframework.validation.Validator;

import mybootapp.model.Product;

@Service
public class ProductValidator implements Validator {

    @Override
    public boolean supports(Class<?> clazz) {
        return Product.class.isAssignableFrom(clazz);
    }

    @Override
    public void validate(Object target, Errors errors) {
        Product product = (Product) target;

        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "name",
                "product.name", "Field␣name␣is␣required.");

        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "description",
                "product.description", "Field␣description␣is␣required.");

        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "type",
                "product.type", "Field␣type␣is␣required.");

        if (!(product.getPrice() > 0.0)) {
            errors.rejectValue("price", "product.price.too.low",
                    "Price␣too␣low");
        }
    }

}
```

Pour l'utiliser, il suffit de modifier le contrôleur comme suit :

```java
@Autowired
ProductValidator validator;

@PostMapping("/edit")
public String saveProduct(@ModelAttribute Product p, BindingResult result) {
    validator.validate(p, result);
    if (result.hasErrors()) {
        return "productForm";
    }
    repo.save(p);
    return "redirect:list";
}
```

4.7 Traduire les messages de validation

Pour l'instant les messages d'erreurs sont affichés en anglais. Nous pouvons les traduire automatiquement en délocalisant ces messages dans des fichiers de ressources.

Commencez par créer dans le répertoire des ressources (`src/main/resources`) le fichier `product.properties` :

```
product.name = Name is required!
product.description = Description is required!
product.type = Type is required!
product.price.too.low = Price is too low!
```

puis le fichier `product_fr_FR.properties` :

```
product.name = Le nom est requis
product.price.too.low = Le prix est trop bas !
```

Tous les messages sont donnés en anglais. Certains sont en français.

Pour exploiter ces ressources, nous allons les charger en ajoutant dans la classe `Starter` la création d'un nouveau service :

```
@Bean("messageSource")
public MessageSource messageSource() {
    var r = new ReloadableResourceBundleMessageSource();
    r.setBasenames("classpath:product", "classpath:messages");
    return r;
}
```

Nous pouvons simplifier notre classe de validation en supprimant les messages. Elle devient :

```
package mybootapp.web;

import org.springframework.stereotype.Service;
import org.springframework.validation.Errors;
import org.springframework.validation.ValidationUtils;
import org.springframework.validation.Validator;

import mybootapp.model.Product;

@Service
public class ProductValidator implements Validator {

    @Override
    public boolean supports(Class<?> clazz) {
        return Product.class.isAssignableFrom(clazz);
    }

    @Override
    public void validate(Object target, Errors errors) {
        Product product = (Product) target;

        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "name",
                "product.name");

        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "description",
                "product.description");

        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "type",
                "product.type");

        if (!(product.getPrice() > 0.0)) {
            errors.rejectValue("price", "product.price.too.low");
        }
    }

}
```

Travail à faire : **mais plus tard**. Suivez ce tutoriel pour changer la langue sur la demande de l'utilisateur [a].

a. https://www.mkyong.com/spring-mvc/spring-mvc-internationalization-example/

4.8 Traiter des champs complexes

Nous venons de le voir, Spring MVC traite parfaitement les champs qui correspondent à un type de base (entier, flottant, chaîne, etc.). Nous allons maintenant nous occuper des champs complexes.

Commençons par définir une classe pour représenter le numéro de série d'un produit (une lettre suivie d'un entier entre 1000 et 9999) :

```
package mybootapp.model;

import jakarta.persistence.Column;
import jakarta.persistence.Embeddable;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Embeddable
@Data
@NoArgsConstructor
@AllArgsConstructor
public class ProductCode {

    @Column(name = "code_base")
    String base;

    @Column(name = "code_number")
    int number;

}
```

et ajoutons ce code à notre produit :

```
...

@Entity
@Data
public class Product {

    ...

    @Embedded
    private ProductCode code;

    ...

}
```

Faites ensuite les modifications suivantes :

- dans `ProductController` associez le code `A1000` au premier produit et `B2000` au deuxième.
- ajoutez le code suivant à votre formulaire :

```
<div class="form-group my-1">
    <label for="code">Code:</label>
    <form:input path="code.base" class="form-control"/>
    <form:input path="code.number" class="form-control"/>
    <form:errors path="code" cssClass="alert alert-warning"
        element="div" />
</div>
```

- Modifiez le validateur en conséquence en ajoutant

```
ProductCode code = product.getCode();
if (code != null) {
    if (!code.getBase().matches("[A-Z]")) {
        errors.rejectValue("code", "product.code.base");
    }
    if (!(code.getNumber() >= 1000 && code.getNumber() <= 9999)) {
        errors.rejectValue("code", "product.code.number");
    }
}
```

- Ajoutez des messages d'erreurs pour `product.code.base` et `product.code.number`.

Vous devez maintenant être capable d'éditer les deux parties du numéro de série.

Une autre solution consiste à fournir une classe d'adaptation (un éditeur dans la terminologie des JavaBeans) qui est capable de transformer un code en chaîne et vice-versa. En voici un exemple :

```
package mybootapp.web;

import java.beans.PropertyEditorSupport;

import mybootapp.model.ProductCode;

class ProductCodeEditor extends PropertyEditorSupport {

    @Override
    public String getAsText() {
        Object o = this.getValue();
        if (o instanceof ProductCode) {
            ProductCode c = (ProductCode) o;
            return c.getBase() + "" + c.getNumber();
        }
        return super.getAsText();
    }

    @Override
    public void setAsText(String text) throws IllegalArgumentException {
        try {
            String base = text.substring(0, 1);
            int number = Integer.parseInt(text.substring(1));
            ProductCode c = new ProductCode(base, number);
            super.setValue(c);
        } catch (Exception e) {
            throw new IllegalArgumentException("Bad code format");
        }
    }

}
```

Il suffit maintenant d'indiquer au contrôleur que nous disposons de cette classe. Pour ce faire nous allons lui adjoindre une méthode annotée par `InitBinder` :

```
@InitBinder
public void initBinder(WebDataBinder b) {
    b.registerCustomEditor(ProductCode.class, new ProductCodeEditor());
}
```

Le formulaire peut devenir :

```
<div class="form-group my-1">
    <label for="code">Code:</label>
    <form:input path="code" class="form-control" />
    <form:errors path="code" cssClass="alert alert-warning"
        element="div" />
</div>
```

Testez le bon fonctionnement de cette nouvelle version.

Note : Si le code renseigné dans le formulaire est incorrect, nous voyons apparaître dans le formulaire l'exception `IllegalArgumentException` générée par l'adaptateur. Ce n'est évidemment pas souhaitable. Vous pouvez préparer votre propre message en ajoutant :

Ajout dans `product.properties`

```
typeMismatch.mybootapp.model.ProductCode=Bad product code
```

Ajout dans `product_fr_FR.properties`

```
typeMismatch.mybootapp.model.ProductCode=Code produit incorrect
```

5 Validation des JavaBean dans JEE 6/7/8

Une nouvelle spécification (JSR303) nous permet d'exprimer les contraintes sur les propriétés par des annotations (plus d'information dans la documentation JEE 6 [4] et dans la JavaDoc [5]).

5.1 Mise en oeuvre

Nous avons déjà prévu, dans le fichier `pom.xml`, les dépendances pour utiliser l'implantation d'Hibernate de la spécification JSR 303.

Continuons en utilisant les annotations dans nos POJOs. Voila la nouvelle version de la classe `product` :

4. http://docs.oracle.com/javaee/6/tutorial/doc/gircz.html
5. http ://docs.oracle.com/javaee/6/api/javax/validation/constraints/package-summary.html

```java
package mybootapp.model;

import jakarta.persistence.Basic;
import jakarta.persistence.Embedded;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.Id;

import jakarta.validation.Valid;
import jakarta.validation.constraints.Min;
import jakarta.validation.constraints.NotEmpty;
import jakarta.validation.constraints.NotNull;
import jakarta.validation.constraints.Size;

import lombok.Data;

@Entity
@Data
public class Product {

    @Id
    @GeneratedValue
    private Long number;

    @NotEmpty
    @Size(min = 1, message = "Le nom est obligatoire")
    @Basic
    private String name;

    @NotNull
    @Min(value = 1, message = "Le prix est trop bas")
    @Basic
    private Double price;

    @NotEmpty(message = "La description est obligatoire")
    @Size(min = 1, max = 100, message = "Entre 1 et 200 caractères")
    @Basic
    private String description;

    @NotEmpty
    @Size(min = 1, message = "Le type doit être renseigné")
    @Basic
    private String type;

    @Valid
    @Embedded
    private ProductCode code;

}
```

et

```java
package mybootapp.model;

import jakarta.persistence.Column;
import jakarta.persistence.Embeddable;

import jakarta.validation.constraints.Max;
import jakarta.validation.constraints.Min;
import jakarta.validation.constraints.NotNull;
import jakarta.validation.constraints.Pattern;
import jakarta.validation.constraints.Size;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Embeddable
@Data
@NoArgsConstructor
@AllArgsConstructor
public class ProductCode {

    @NotNull
    @Size(min = 1, max = 1)
    @Pattern(regexp = "[A-Z]", message = "Le code doit débuter par une majuscule")
    @Column(name = "code_base")
    String base;

    @Min(value = 1000, message = "Le numéro doit être >= à 1000")
    @Max(value = 9999, message = "Le numéro doit être <= à 9999")
    @Column(name = "code_number")
    int number;

}
```

Nous allons ajouter l'annotation `@Valid` dans le contrôleur :

```java
...

@PostMapping("/edit")
public String saveProduct(@ModelAttribute @Valid Product p, BindingResult result) {
    validator.validate(p, result);
    if (result.hasErrors()) {
        return "productForm";
    }
    repo.save(p);
    return "redirect:list";
}

...
```

Travail à faire : Testez le résultat. Vous devez vous retrouver avec les messages en provenance du validateur plus les nouveaux messages en provenance des annotations. Vous pouvez maintenant vider votre validateur manuel. La classe de validation reste **utile** pour les **validations métier** (par exemple pour vérifier que telle propriétés doit être supérieure à telle autre ou pour vérifier l'unicité d'une données en base).

5.2 Contrôler les messages

Si vous souhaitez contrôler les messages de validation, vous pouvez (par exemple pour le nom du produit), ajouter un service Spring pour faire le lien entre la validation et vos messages (la méthode `getValidator` est définie dans l'interface `WebMvcConfigurer` [6])

Ajout dans `Starter.java`

```
@Override
public LocalValidatorFactoryBean getValidator() {
    var factory = new LocalValidatorFactoryBean();
    factory.setValidationMessageSource(messageSource());
    return factory;
}
```

Vous pourrez ainsi faire référence aux messages dans les annotations de validation :

Modification de `Product.java`

```
...

public class Product {

    ...

    @NotEmpty
    @Size(min = 1, message="{product.name}")
    // avant @Size(min = 1, message = "Le nom est obligatoire")
    @Basic
    private String name;

    ...

}
```

5.3 Créer ses propres contraintes

Le mécanisme de validation peut facilement être étendu pas ajout de contraintes spécifiques. Nous allons créer une contrainte `Bye` qui va vérifier la présence de ce mot dans un champ. Pour ce faire, commencez par créez l'annotation `Bye` :

6. https ://docs.spring.io/spring/docs/6.0.14/javadoc-api/org/springframework/web/servlet/config/annotation/WebMvcConfigurer.html

```java
package mybootapp.web;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import jakarta.validation.Constraint;
import jakarta.validation.Payload;

@Target({ ElementType.METHOD, ElementType.FIELD, ElementType.ANNOTATION_TYPE })
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = ByeConstraintValidator.class)
@Documented
public @interface Bye {

    String message() default "Il␣manque␣le␣'bye'";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

}
```

Puis continuez en créant la classe de validation `ByeConstraintValidator` :

```java
package mybootapp.web;

import jakarta.validation.ConstraintValidator;
import jakarta.validation.ConstraintValidatorContext;

public class ByeConstraintValidator implements ConstraintValidator<Bye, String> {

    @Override
    public void initialize(Bye arg0) {
    }

    @Override
    public boolean isValid(String arg0, ConstraintValidatorContext arg1) {
        if (arg0.contains("bye"))
            return true;
        return false;
    }

}
```

Modifez ensuite la classe `Product` pour utiliser cette annotation :

```java
public class Product {

    ...

    @NotEmpty(message = "La␣description␣est␣obligatoire")
    @Size(min = 1, max = 100, message = "Entre␣1␣et␣200␣caractères")
    @Bye
    @Basic
    private String description;

    ...

}
```

Travail à faire : Vérifiez son bon fonctionnement !

6 C'est fini

À bientôt.