1. Présentation du framework "Spring MVC"

"Spring Web MVC" est une partie optionnelle du framework spring servant à gérer la logique du design pattern "MVC" dans le cadre d'une intégration "spring" .

"Spring MVC" est à voir comme un petit framework java/web (pour le coté serveur) qui peut être soit vu comme une alternative à Struts2 ou JSF2 soit être vu comme un petit framework web complémentaire à Struts2 ou JSF2.

Dépendances maven nécessaires :

```
<dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-web</artifactId>
        <version>${spring.version}</version>
</dependency>

<dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>${spring.version}</version>
</dependency>
```

Configuration :

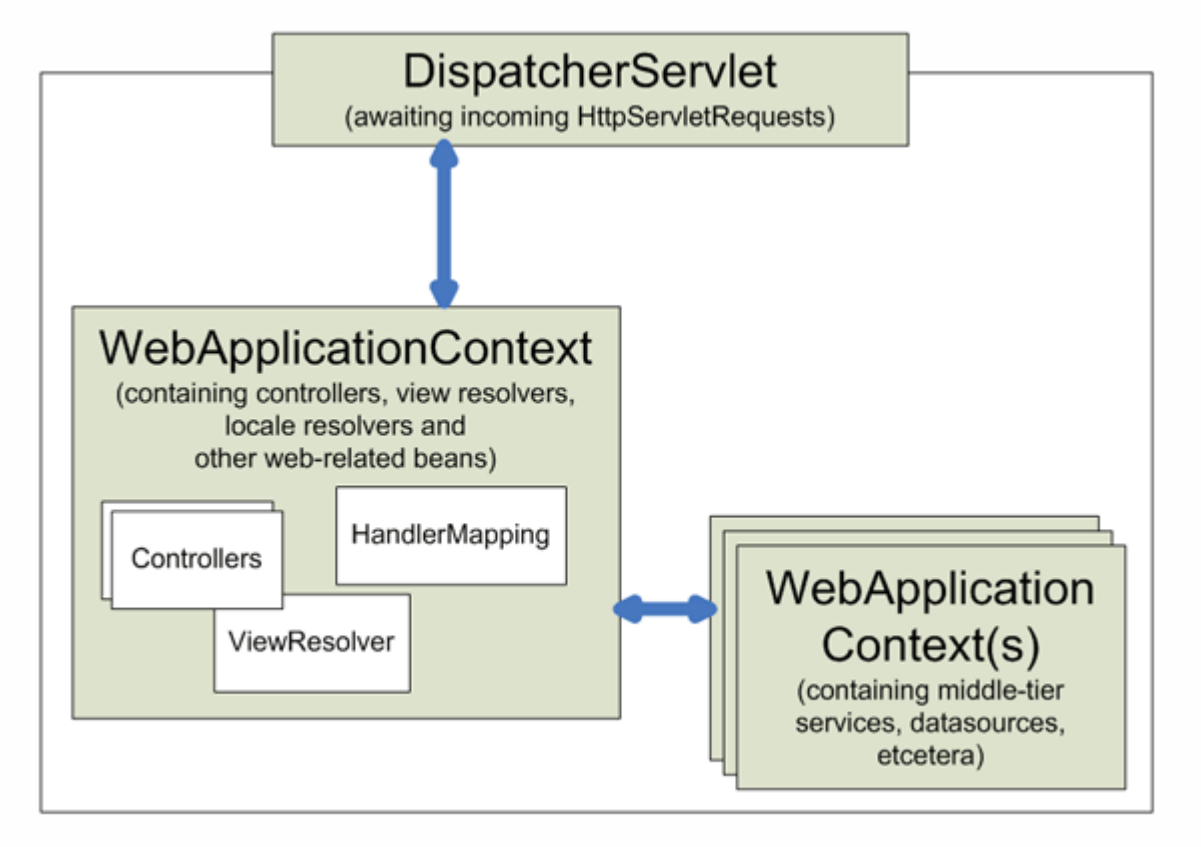
WEB-INF/**web.xml**

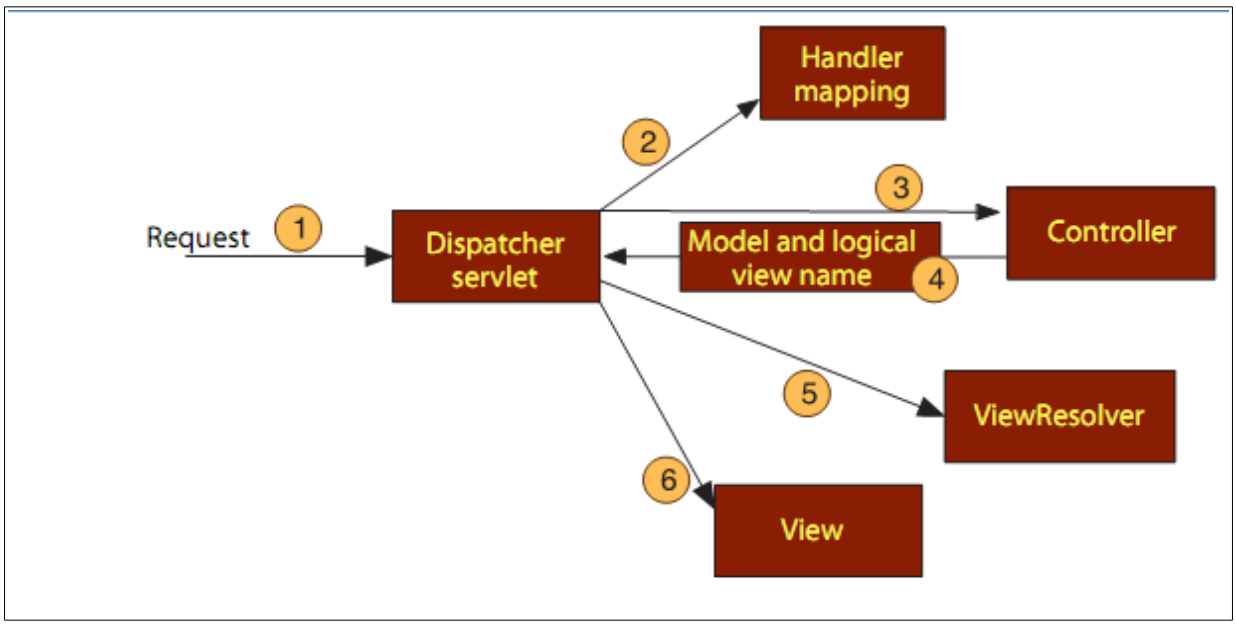
```
...
    <servlet>
            <servlet-name>mvc-dispatcher</servlet-name>
            <servlet-class>
                org.springframework.web.servlet.DispatcherServlet
            </servlet-class>
        <init-param>
             <param-name>contextConfigLocation</param-name>
             <param-value>/WEB-INF/mvc-config.xml</param-value>
        </init-param>
            <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
            <servlet-name>mvc-dispatcher</servlet-name>
            <url-pattern>/mvc/*</url-pattern>
    </servlet-mapping>

    <context-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>/WEB-INF/classes/spring-mvc.xml,...</param-value>
    </context-param>

    <listener>
            <listener-class>
                org.springframework.web.context.ContextLoaderListener
            </listener-class>
    </listener>
```

WEB-INF/**mvc-config.xml** (spring mvc)

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>
        <context:component-scan      base-package="tp.web.mvc.controller"/>
        <mvc:annotation-driven />


          <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
                <!-- Example: a logical view name of 'showMessage' is mapped to
                                '/WEB-INF/view/showMessage.jsp' -->
                <property name="prefix" value="/WEB-INF/view/"/>
                <property name="suffix" value=".jsp"/>
          </bean>
</beans>
```

Fonctionnement





les étapes fondamentales sont :
1.Le DispatcherServlet reçoit une requête dont l'URI-pattern est '/xy.htm'
2.Le DispatcherServlet consulte son ***Handler Mapping*** (Ex : *BeanNameUrlHandlerMapping*) ***pour connaître le contrôleur dont le nom de bean est '/xy.htm'*** .
3.Le DispatcherServlet dispatche la requête au contrôleur identifié (Ex : XyPageController)
4.**Le contrôleur retourne au DispatcherServlet un objet de type ModelAndView possédant comme paramètre au minimum le nom logique de la vue à renvoyer (ou bien un objet Model plus le nom logique de la vue sous forme de String depuis la version 3)**
5.Le DispatcherServlet consulte son ***View Resolver*** lui permettant de trouver la vue dont le nom logique est 'xy' ou 'zzz'. Ici le type de View Resolver choisit est *InternalResourceViewResolver*.
6.Le DispatcherServlet "forward" ensuite la requête à la **vue associé (page jsp ou ...)**

Exemple simple de contrôleur :

```
package tp.web.mvc.controller;
...
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.AbstractController;

@Controller
public class HelloWorldController extends AbstractController{

	@Override
	@RequestMapping("/helloWorld")
	protected ModelAndView handleRequestInternal(HttpServletRequest request,
			HttpServletResponse response) throws Exception {
		return new ModelAndView("showMessage","message","helloWorld");
					//viewName	, nameData  , valueData
	}
}
```

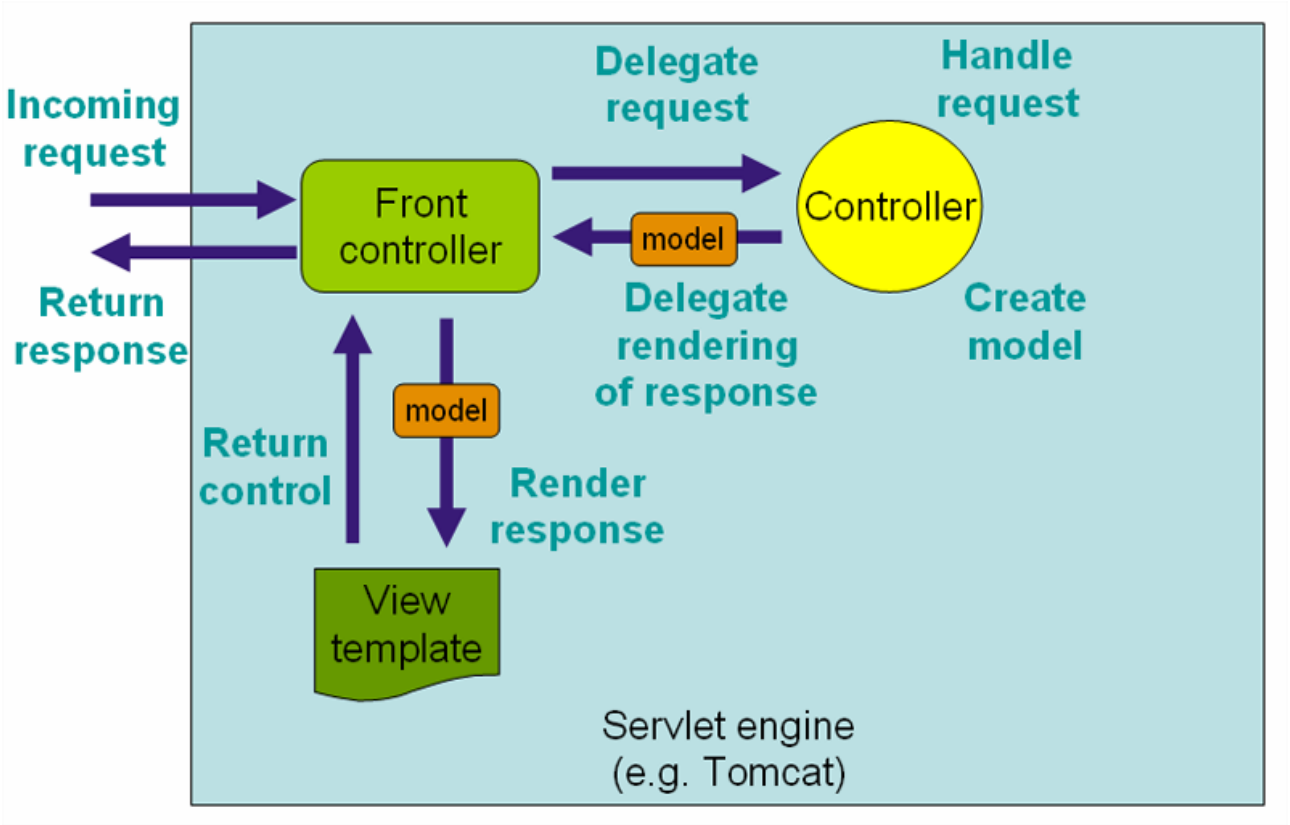
et éventuellement avec cette configuration xml (si pas d'annotation "@Controller" ni "@RequestMapping") :

```
<bean  class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>

  <bean name="/helloWorld"    class="tp.web.mvc.controller.HelloWorldController" />
  <bean name="/url2"    class="tp.web.mvc.controller.ForUrl2Controller" />
```

ou bien (plus simplement sans héritage depuis la v3) :

```
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class HelloWorldController {

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

}
```

Au niveau de showMessage.jsp, l'affichage de message pourra être effectué via ${message}.



2. éléments essentiels de Spring web MVC

2.1. éventuelle génération directe de la réponse HTTP

```
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller //but not "@Component" for spring web controller
@RequestMapping("/app")
public class WelcomeCtrl {

        @RequestMapping("/hello")
          @ResponseBody //si @ResponseBody , génération directe de la réponse ,
                        // sinon viewResolver (mvc-config.xml)
          String say_hello() {
             return "Hello World!";
          }
}
```

2.2. @RequestParam (accès aux paramètres HTTP)

conversion.jsp

```
… <form action="doConversion" method="GET_ou_POST">
          source: <select name="source" >
                  <c:forEach var="d" items="${allDevises}">
                       <option  value="${d.monnaie}">${d.monnaie}</option>
                  </c:forEach>
                 </select>  <br/>
          cible: <select name="cible" > ... </select> <br/>
          montant: <input name="montant"  value="${montant}"/> <br/>
          <input  type="submit" value="convertir" /> <br/>
        </form>
        sommeConvertie=<b>${sommeConvertie}</b>   ...
```

```
@RequestMapping("/doConversion")
public String doConversion(Model model,@RequestParam(name="montant")double montant,
                                       @RequestParam(name="source")String monnaieSrc,
                                       @RequestParam(name="cible")String monnaieDest)
{   ….
   model.addAttribute("sommeConvertie",
                      gestionDevises.convertir(montant, monnaieSrc, monnaieDest));
   return "conversion";
}
```

2.3. @ModelAttribute

Pour spécifier un attribut du modèle on peut appeler ***model.addAttribute("attrName", attrVal);*** au sein d'une méthode préfixée par @RequestMapping .

Une autre solution consiste à coder une méthode addXyModelAttribute() préfixée par @ModelAttribute("attrName").

Exemple :

```
@ModelAttribute("conv")
        public ConversionForm addConvAttributeInModel() {
                return new ConversionForm();
        }
```

Le framework "spring mvc" va alors appeler automatiquement (*) toutes les méthodes préfixées par

@ModelAttribute pour initialiser certains attributs du modèle avant de déclencher les méthodes préfixées par *@RequestMapping*.

L'appel n'est effectué que pour initialiser la valeur d'un attribut n'existant pas encore (pas d'écrasement des valeurs en session ni des valeurs saisies via <form:form …./>)

Une méthode préfixée par ***@ModelAttribute*** peut éventuellement avoir un paramètre préfixé par ***@RequestParam(name="numCli",required=true_or_false)*** mais elle n'a pas le droit de retourner une valeur "null" pour un attribut du modèle .

Variante syntaxique (en void et avec model) pour de multiples initialisations :

```
@ModelAttribute
        public void addAttributesInModel(Model model){
  model.addAttribute("xx", new Cxx());
  model.addAttribute("yy", new Cyy());
}
```

Autre Exemple :

```
@Controller //but not "@Component" for spring web controller
//@Scope(value="singleton")//by default
@RequestMapping("/devises")
public class DeviseListCtrl {

        @Autowired //ou @Inject
        private GestionDevises gestionDevises;

        private List<Devise> listeDevises = null; //cache


        @PostConstruct
```

```
        private void loadListeDevises(){
                if(listeDevises==null)
                        listeDevises=gestionDevises.getListeDevises();
        }

   @ModelAttribute("allDevises")
        public List<Devise> addAllDevisesAttributeInModel() {
            return listeDevises;
        }

         @RequestMapping("/liste")
           public String toDeviseList(Model model) {
             //model.addAttribute("allDevises", listeDevises);
             return "deviseList";
           }
}
```

deviseList.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>liste des devises</title>
</head>
<body>
      <h3>liste des devises (spring web mvc)</h3>
      <table border="1" >
      <tr><th>code</th><th>devise</th><th>change</th></tr>
            <c:forEach var="d" items="${allDevises}">
                  <tr><td>${d.codeDevise}</td><td>${d.monnaie}</td>
                      <td>${d.DChange}</td></tr>
            </c:forEach>
      </table>
      <hr/>
      <a href="../app/to_welcome">retour page accueil</a> <br/>
</body>
</html>
```

<u>Accès à un attribut pour effectuer une mise à jour</u>:

```
@RequestMapping("/info")
   public String toInfosClient(Model model) {
      //mise à jour du telephone du client 0L (pour le fun / la syntaxe):
                Client cli = (Client) model.asMap().get("customer");
                if(cli!=null && cli.getNumero()==0L)
                        cli.setTelephone("0102030405");
      return "infosClient";
   }
```

2.4. @SessionAttributes

```
@Controller
//@Scope(value="singleton")//by default
@RequestMapping("/client")
@SessionAttributes( value={"customer"} )
   //noms des "modelAttributes" qui sont EN PLUS récupérés/stockés
   // en SESSION HTTP au niveau de la page de rendu
   // --> visibles en requestScope ET en sessionScope
public class ClientCtrl {


   //NB: @SessionAttributes et @ModelAttribute sont gérés avant @RequestMapping


        @ModelAttribute("customer") //NB: cette méthode n'est pas appelée/déclenchée
                        //si "customer" est déjà présent en session (et par copie) dans le modèle
        public Client addCustomerAttributeInModel() {
                        return new Client(0L,null,null) ;
        }
}
```

Mettre fin à une session http:

```
@RequestMapping("/endSession")
   public String endSession(Model model,HttpSession session) {
      if(model.containsAttribute("customer"))
                model.asMap().remove("customer");
      session.invalidate();
      return "infosClient";
   }
```

2.5. tags pour formulaires (form:form , form:input , …)

Spring-mvc offre une bibliothèque de tags permettant de simplifier la structuration d'une page JSP comportant un formulaire (à saisir , à valider , ….).

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

Ces nouvelles balises préfixées par *form:* s'utilisent quasiment de la même façon que les balises standards HTML (path="nomPropJava" à la place de name="nomParamHttp").

La principale valeur ajoutée des balises préfixées par *form:* consiste dans les liaisons automatiques entre certaines propriétés d'un objet java et les champs d'un formulaire.

Les balises <form:input ...> , <form:select ….> doivent être imbriquées dans <form:form >.

La balise principale d'un formulaire <***form:form*** *action="actionXY"* ***modelAttribute****="beanName" method="POST" > ... <form:form>* … comporte un attribut clef **modelAttribute** qui doit correspondre à un nom de "modelAttribute" lui même associé à un **objet java comportant toutes les données du formulaire à soumettre**.

Autrement dit , form:form ne fonctionne correctement que si la classe du sous-contrôleur est structurée avec au moins un "@ModelAttribute" (existant dès le départ , pas "null") dont le type correspond à une classe souvent spécifique au formulaire (ex : "UserForm" , "OrderForm" , ….) .

Exemple:

```
public class ConversionForm {
	private Double montant;
	private String monnaieSrc;
	private String monnaieDest;

	public ConversionForm(){
		monnaieSrc="dollar";
		monnaieDest="dollar"; //par défaut (dans formulaire avant saisies)
	}
	//+ get/set
}
```

```
@Controller
//@Scope(value="singleton")//by default
@RequestMapping("/devises")
public class DeviseListCtrlV2 {
…
   //pour modelAttribute="conv" de  form:form
   @ModelAttribute("conv")
	public ConversionForm addConvAttributeInModel() {
		return new ConversionForm();
	}
…
}
```

L'attribut path="..." des sous balises <form:input ...> , <form:select ….> font alors référence aux propriétés de l'objet java (en lecture/écriture , get/set) .

NB: <form:form …> gère (génère) automatiquement le champ caché **_csrf** attendu par **spring-security** . *Exemple* : <input type="hidden" name="_csrf" value="8df91b84-74c1-4013-bd44-ede7b00779a2" />) . Ce champ caché correspond au "*Synchronizer Token Pattern*" (que l'on retrouve dans les frameworks web concurrents "Stuts" ou "JSF") : le coté serveur compare la valeur d'un jeton aléatoire stockée en session http avec celle stockée dans un champ caché et refuse de gérer la requête "re-postée" si la comparaison n'est pas réussie.

D'autre part , le terme ***CSRF*** (signifiant "*Cross Site Request Forgery*" correspond à un éventuel problème de sécurité : un site "malveillant" (utilisé en parallèle au sein d'un navigateur) déclenche

automatiquement (via javascript ou autre) des requêtes non voulues (ex : virement monétaire) en utilisant le contexte d'un site à priori de confiance (mais pas assez protégé) .

Avec <form> (au lieu de <form:form>) , il faut insérer nous même le champ suivant au sein du formulaire d'une page ".jsp" :

<input type="hidden" name="${_csrf.parameterName}" value="${_csrf.token}"/>

conversionV2.jsp

```
<form:form action="doConversion" modelAttribute="conv" method="POST">
  source: <form:select path="monnaieSrc" >
                <form:options items="${allDevises}" itemLabel="monnaie"
itemValue="monnaie"/>
          </form:select> <br/>
  cible: <form:select path="monnaieDest" >
                <form:options items="${allDevises}" itemLabel="monnaie" itemValue="monnaie"/>
       </form:select> <br/>
   montant: <form:input path="montant" />
            <form:errors path="montant" cssClass="error"/><br/>
   <input type="submit" value="convertir" /> <br/>
</form:form>
sommeConvertie=<b>${sommeConvertie}</b>
```

conversion de devises

source: dollar
cible: euro
montant: 45.0
convertir
sommeConvertie=37.5

Finalement , au sein du contrôleur , la méthode déclenchée par le formulaire peut s'écrire de la façon suivante:

```
@RequestMapping("/doConversion")
public String doConversion(Model model,@ModelAttribute("conv") ConversionForm conv ) {

model.addAttribute("sommeConvertie",
                        gestionDevises.convertir(conv.getMontant(),
                                                conv.getMonnaieSrc(), conv.getMonnaieDest()));

 return "conversionV2";
}
```

2.6. validation lors de la soumission d'un formulaire

Rappel: la classe de l'objet utilisé en tant que "modelAttribute" au niveau d'un formulaire peut comporter des annotations @Min , @Max , @Size , @NotEmpty , … de l'api normalisée javax.validation .

Exemples :

```
import javax.validation.constraints.Max;
import javax.validation.constraints.Min;

public class ConversionForm {

        @Min(value=0)
        @Max(value=999999)
        private Double montant;
    …
}
```

```
import javax.validation.constraints.Size;
import org.hibernate.validator.constraints.Email;
import org.hibernate.validator.constraints.NotEmpty;

public class Client {
        private Long numero;    private String nom;       private String prenom;

        @NotEmpty(message = "Please enter your address.")
        @Size(min = 4, max = 128, message = "Your address must between 4 and 128 characters")
        private String adresse;
        private String telephone;

        @NotEmpty
        @Email
        private String email;
…
}
```

Il suffit en suite d'ajouter **@Valid** au niveau du paramètre de la méthode associée à la soumission du formulaire pour que spring-mvc tienne compte des contraintes de validation.

D'autre part, le paramètre (facultatif mais conseillé) de type "**BindingResult**" permet de gérer finement les cas d'erreur de validation :

```
@RequestMapping("/doConversion")
public String doConversion(Model model,
                                @ModelAttribute("conv") @Valid ConversionForm conv ,
                                BindingResult bindingResult) {
        if (bindingResult.hasErrors()) {
                // form validation error
                System.out.println("form validation error: " + bindingResult.toString());
        } else {
                 // form input is ok*/
```

```
        model.addAttribute("sommeConvertie", gestionDevises.convertir(conv.getMontant(),
                                                conv.getMonnaieSrc(), conv.getMonnaieDest()));
        }
return "conversionV2";
}
```

conversion de devises

source: dollar
cible: livre
montant: -5.0 **doit être plus grand que 0**
convertir
sommeConvertie=

---

retour page accueil

numero: 0
nom: Therieur
prenom: alex
adresse: ici **Your address must between 4 and 128 characters**
telephone:
email: alex-therieur **Adresse email mal formée**
update

2.7. Compléments pour mise en page

Pour obtenir de belles mises en pages , on pourra coupler "spring-mvc" avec **bootstrap-css** et/ou "tiles" ou "**thymeleaf**" .