



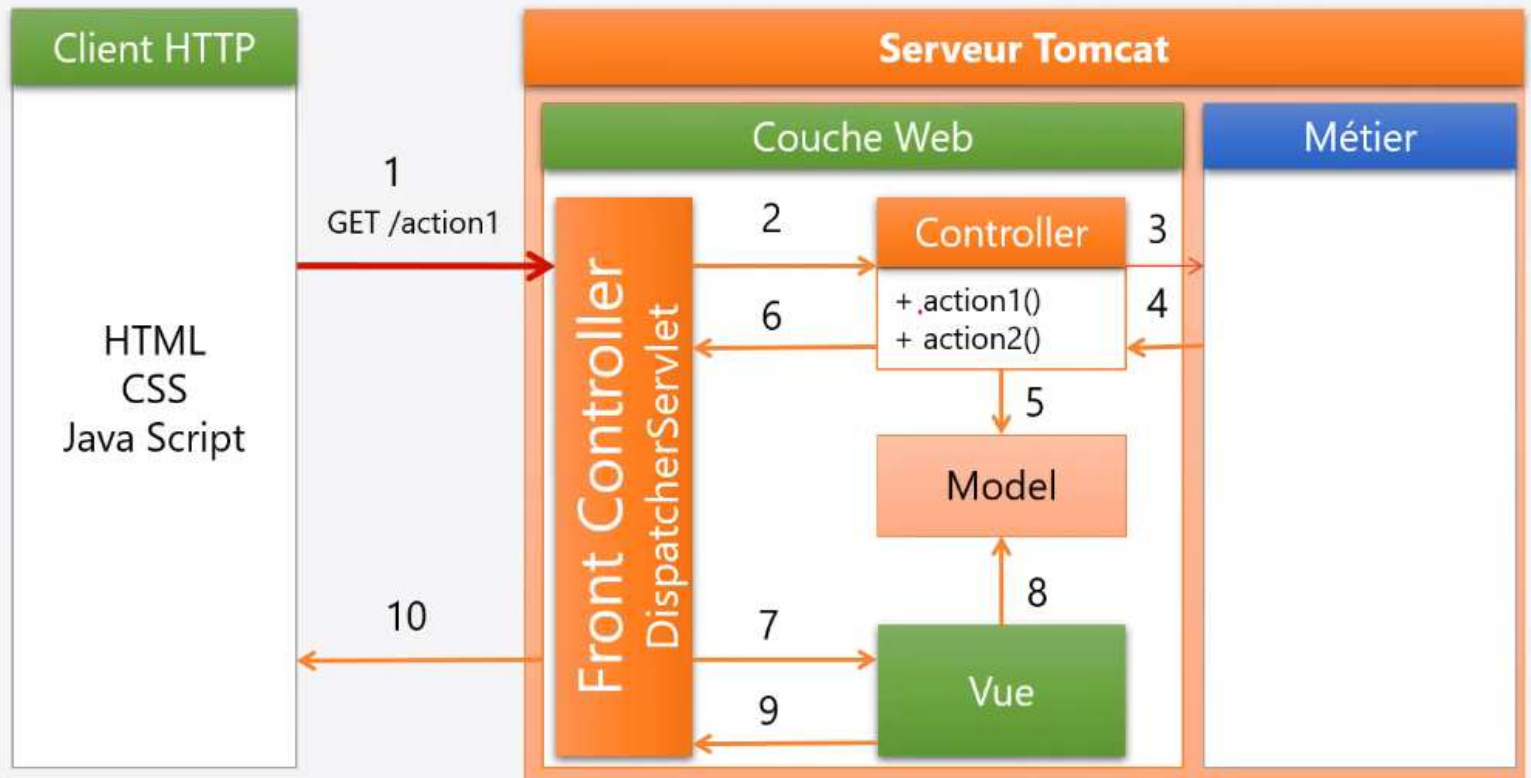
Développement Web JEE

Spring M V C

Architecture Spring MVC

1 -

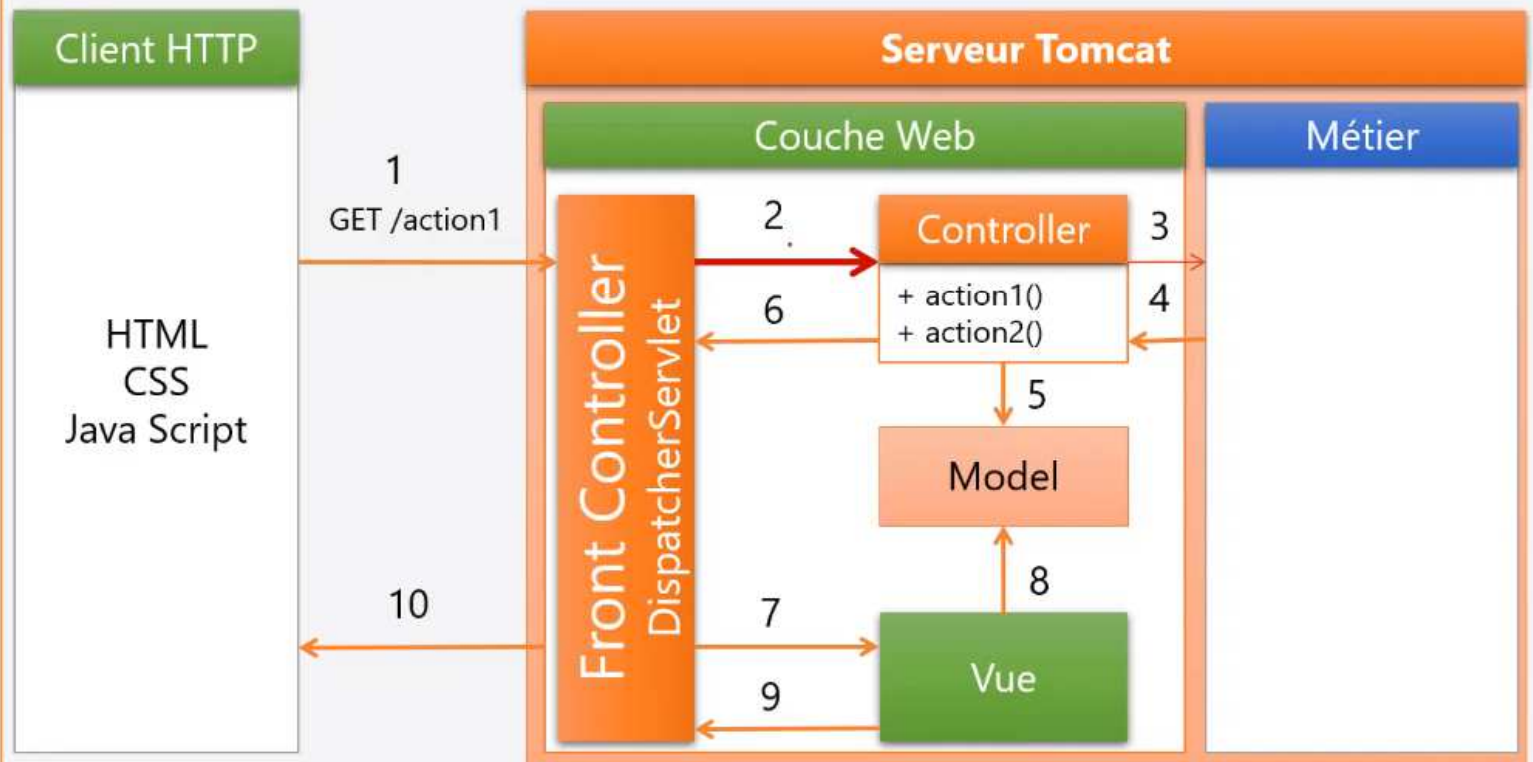
Le client envoie une requête HTTP de type GET ou POST



Architecture Spring MVC

2 -

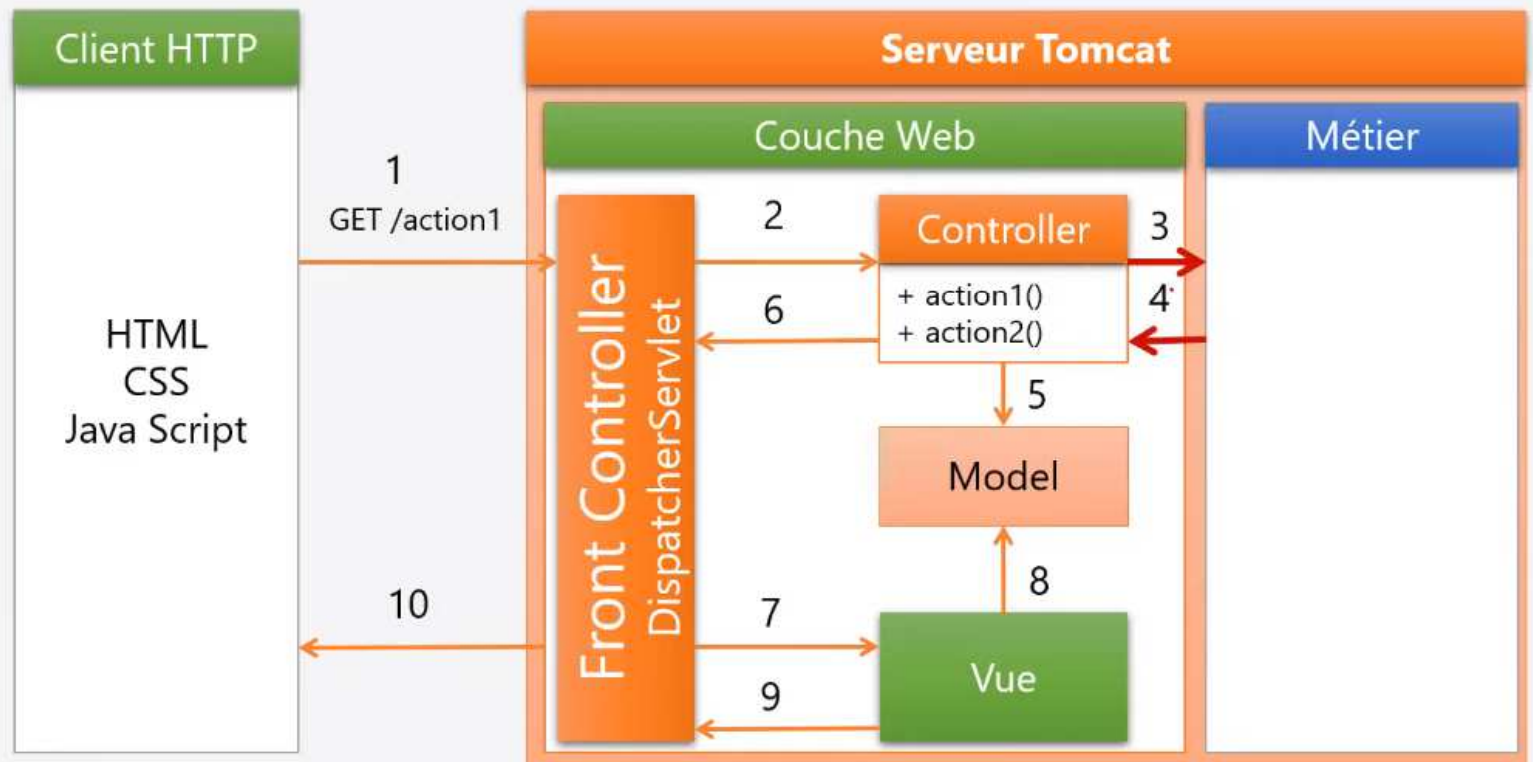
- Toutes les requêtes HTTP sont traitées par un contrôleur frontal fourni par Spring.
- C'est une servlet nommée **DispatcherServlet**.
- Chaque action de l'URL, DispatcherServlet devrait exécuter une opération associée à cette action.
- Cette opération est implémentée dans une classe appelée Controller qui représente un sous contrôleur ou un contrôleur secondaire.



Architecture Spring MVC

3 et 4 –

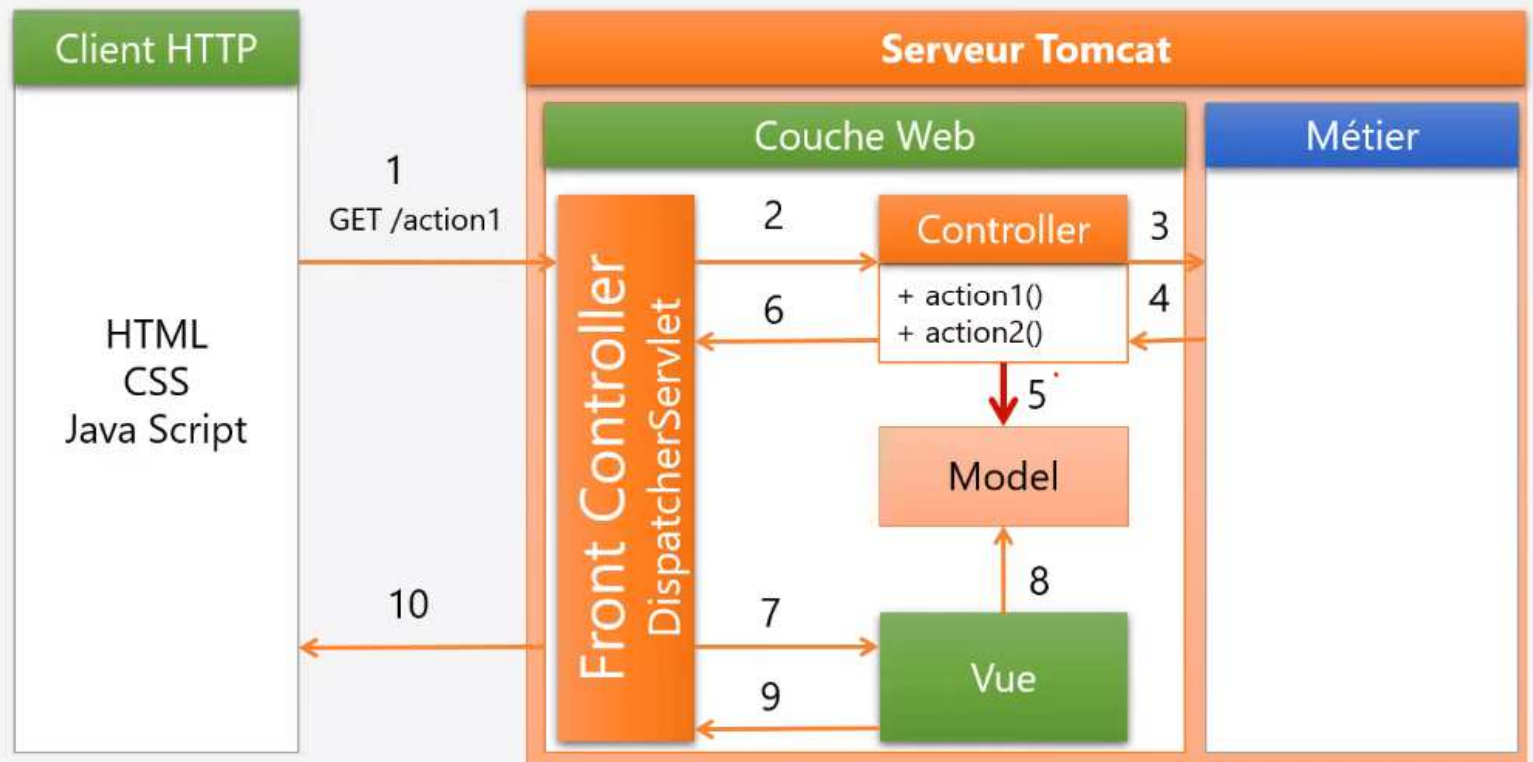
- Le sous contrôleur exécute le traitement associé à l'action en faisant appel à la couche métier et récupère le résultat .



Architecture Spring MVC

5-

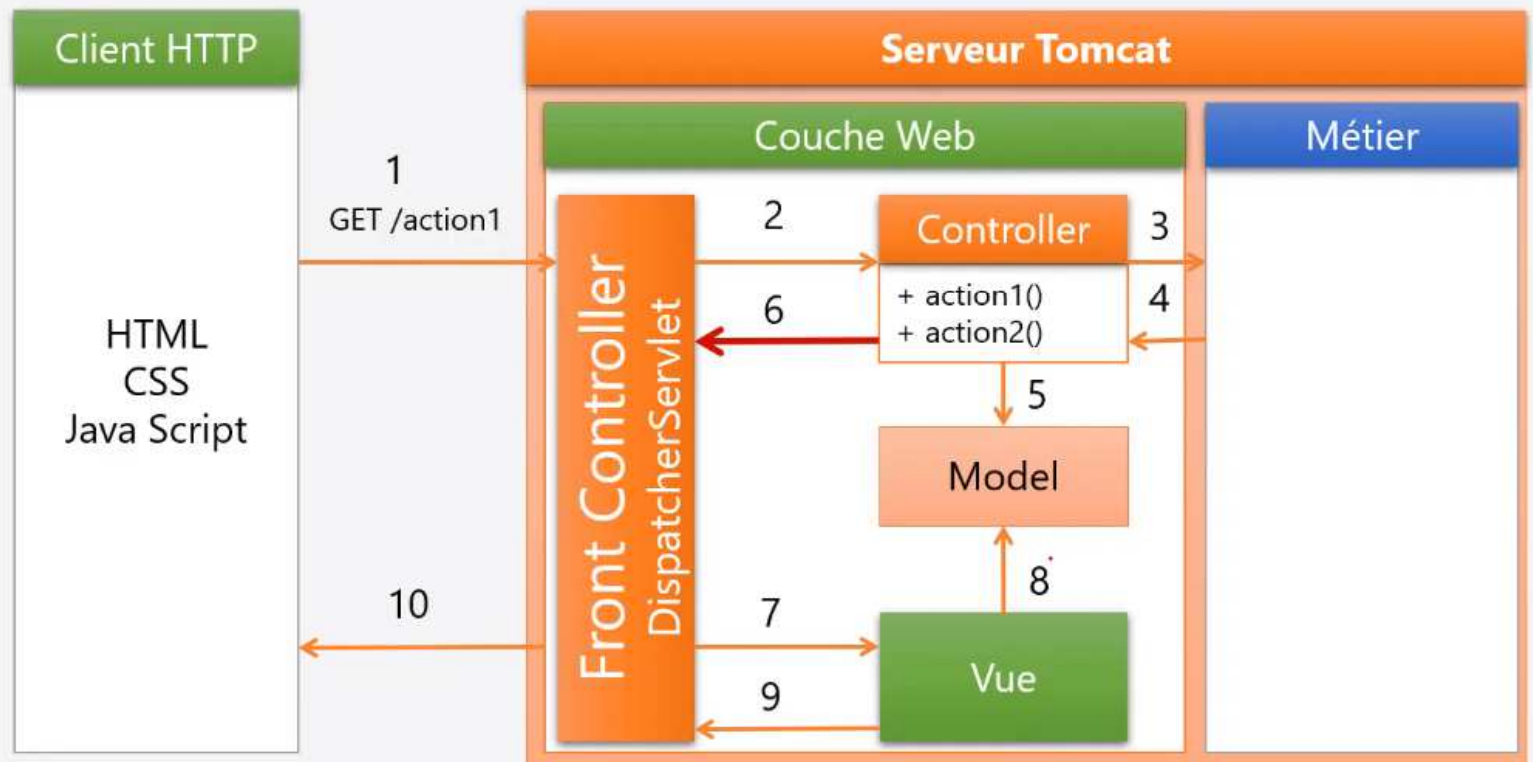
- Le sous contrôleur stocke le résultat dans le modèle fourni par Spring MVC.



Architecture Spring MVC

6-

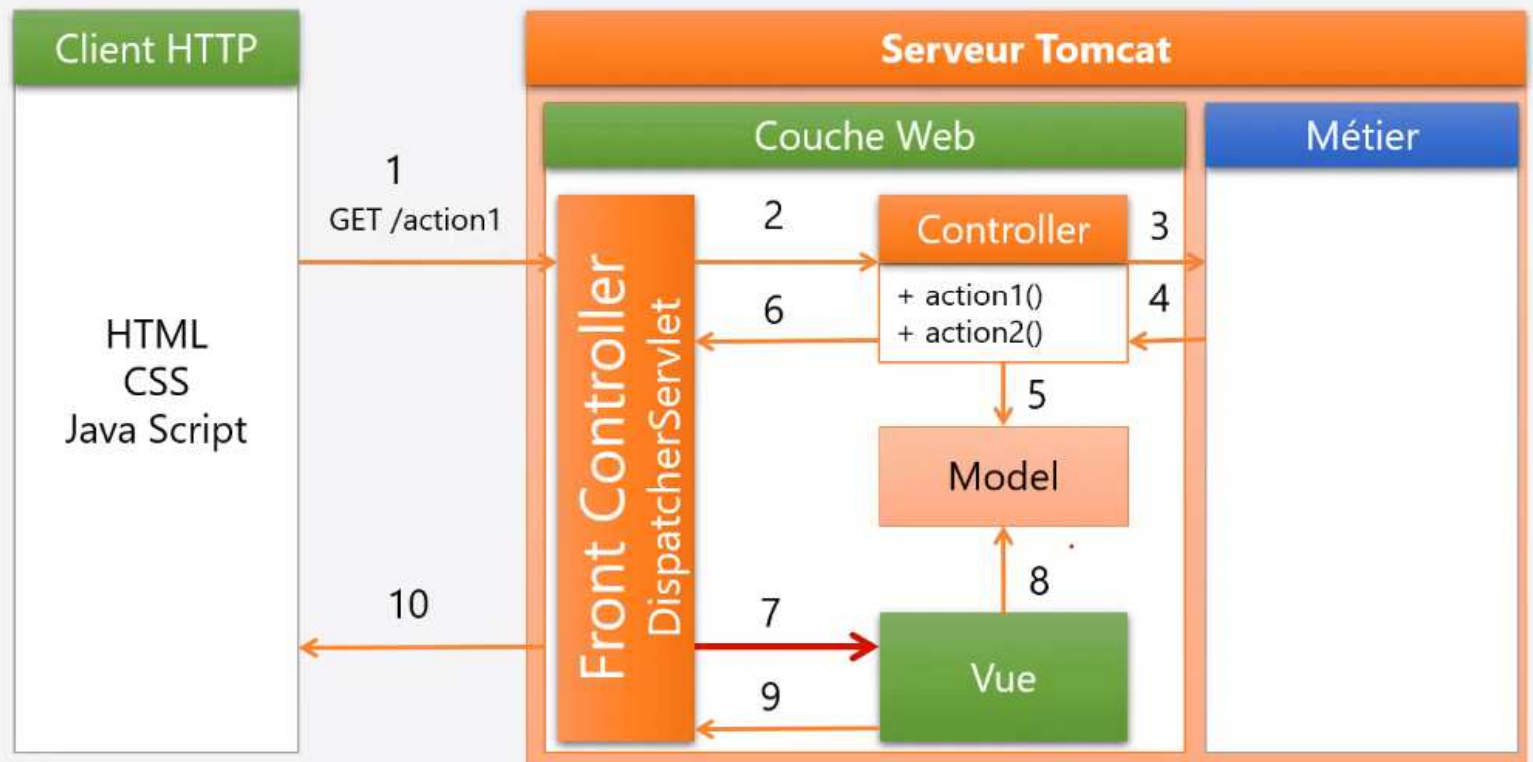
- Le sous contrôleur retourne le nom de la vue et le modèle à DispatcherServlet



Architecture Spring MVC

7-

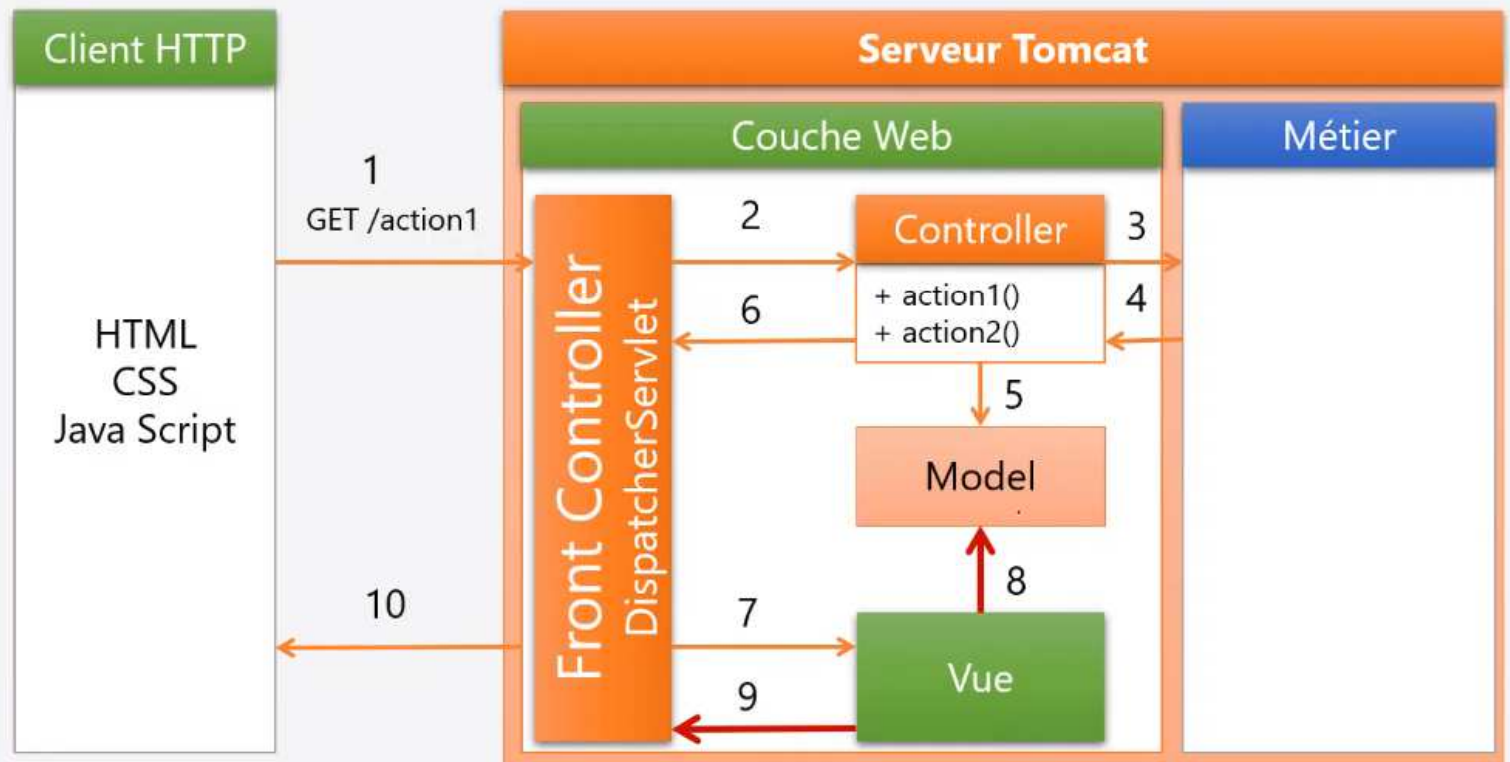
- Le contrôleur frontal DispatcherServlet fait appel à la vue et lui transmet le modèle



Architecture Spring MVC

8 et 9 –

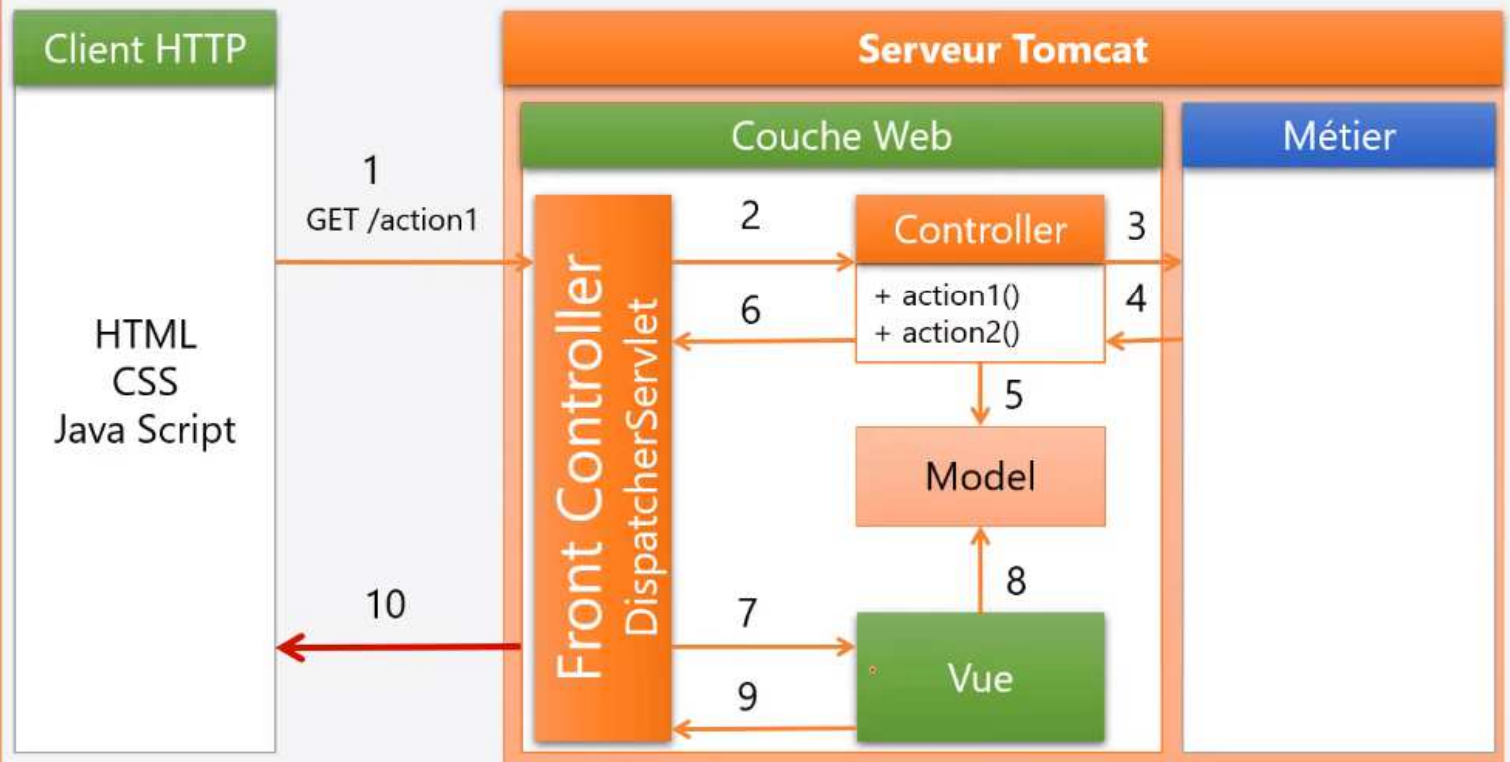
- La vue récupère les résultats à partir du modèle et génère un rendu HTML qui est retourné à DispatcherServlet
- Pour générer du code HTML, on peut utiliser JSP, mais il est déconseillé car il y a mieux : les Moteurs de templates
- Spring MVC offre des moteurs de templates comme **Thymeleaf**, FreeMaker, Mustach, qui permettent de faciliter la génération du code HTML coté serveur



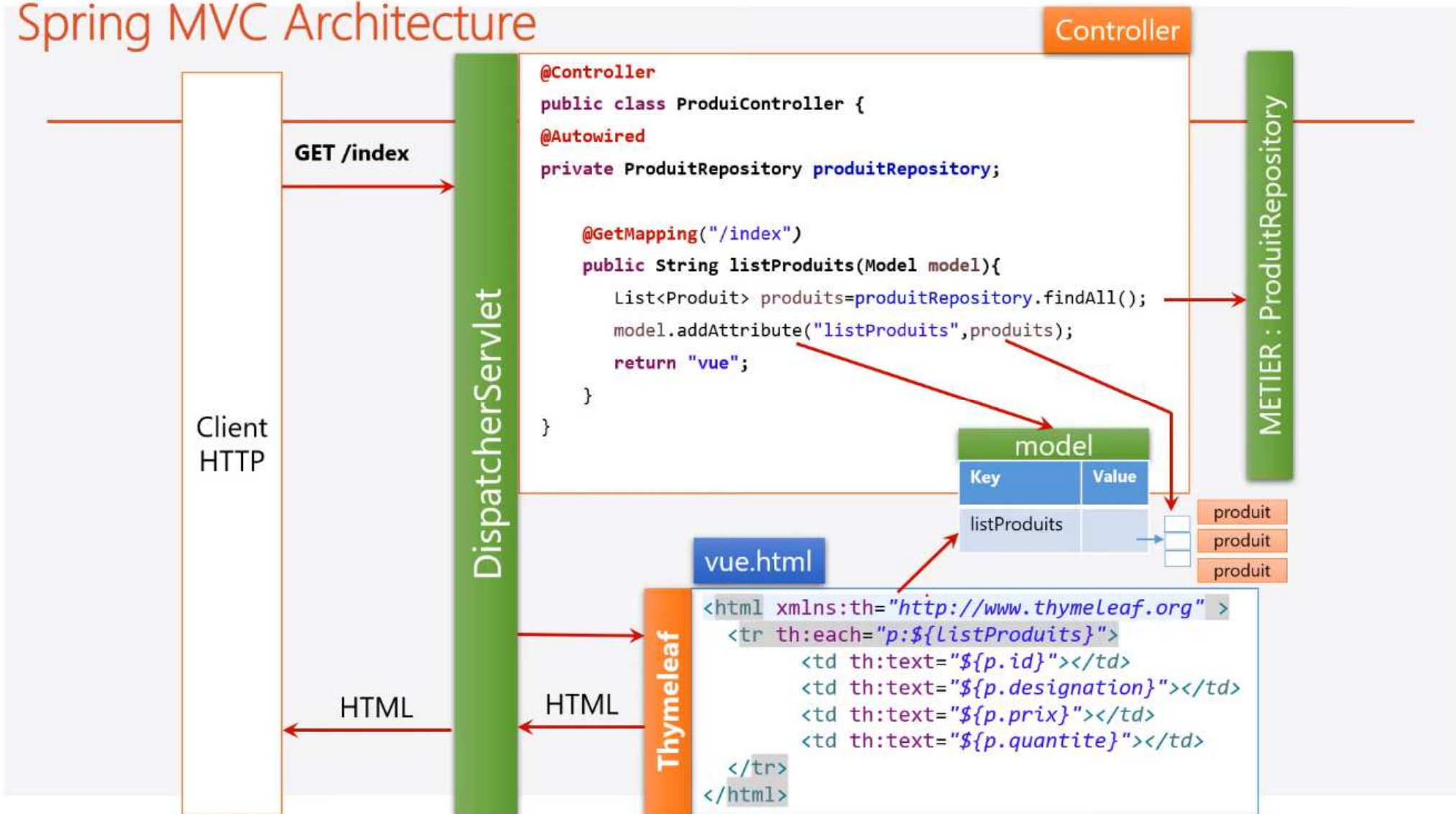
Architecture Spring MVC

10-

- DispatcherServlet envoie la réponse HTTP au client. Cette réponse http contient le code HTML générée par la vue.



Spring MVC Architecture

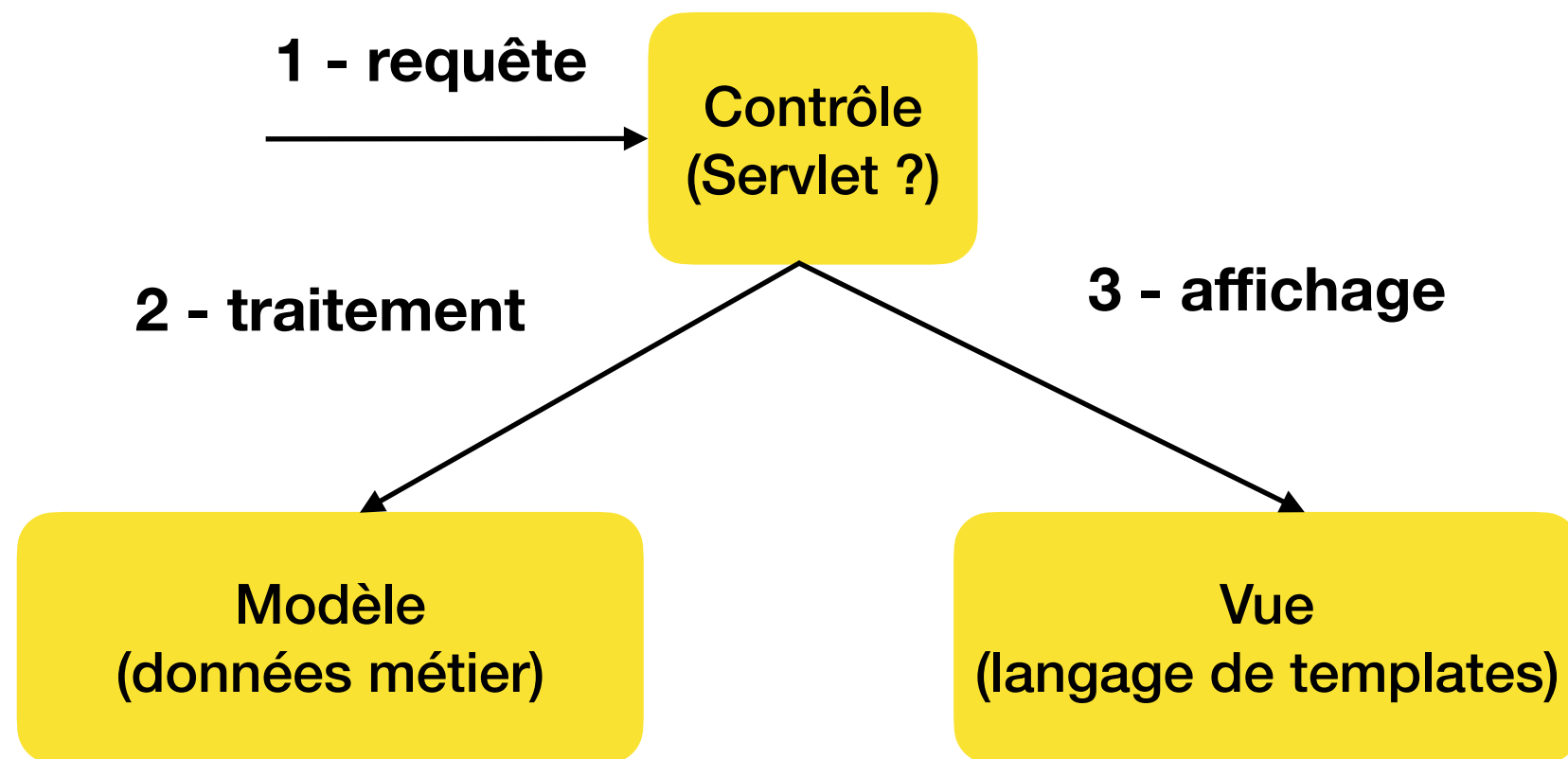


Spring MVC Architecture Rendu Coté Client



Architecture MVC et Web

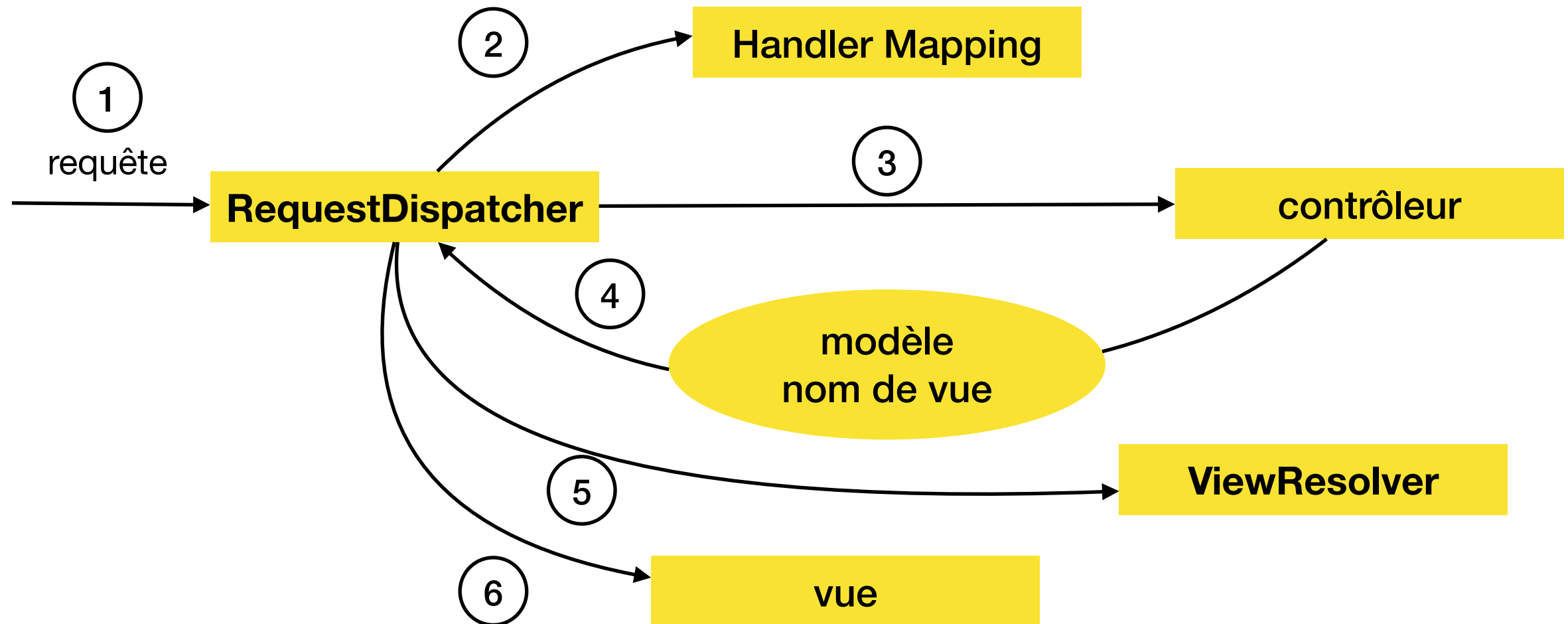
- But: dissocier la visualisation des traitements



Spring MVC

- Un *front controller*, la `DispatcherServlet`, pilote Spring MVC
- Délègue la plupart des traitements à des méthodes placées dans des objets `@Controller`
- L'affichage est généralement pris en charge par des objets « vues », qui tirent leur données d'objets « modèles »
- Tout - ou presque - est modulable.

Spring MVC



D'après *Spring in Action, 3rd Ed. p. 165*

Un exemple simple

```
@Controller
@RequestMapping("/simple")
public class ControleurSimple {

    @RequestMapping(value = "/add", method = RequestMethod.GET)
    public ModelAndView somme(int a, int b) {
        int somme = a + b;
        return new ModelAndView("simpleCalc", "resultat", somme);
    }

    ...
}
```


Un exemple simple

```
@Controller
@RequestMapping("/simple")
public class ControleurSimple {

    @RequestMapping(value = "/add", method = RequestMethod.GET)
    public ModelAndView somme(int a, int b) {
        int somme = a + b;
        return new ModelAndView("simpleCalc", "resultat", somme);
    }

    ...
}
```

@Controller marque la classe comme un contrôleur (sinon, rien ne se passe).
@RequestMapping : les URL associées aux méthodes de la classe commencent par « /simple »

Un exemple simple

```
@Controller
@RequestMapping("/simple")
public class ControleurSimple {

    @RequestMapping(value = "/add", method = RequestMethod.GET)
    public ModelAndView somme(int a, int b) {
        int somme = a + b;
        return new ModelAndView("simpleCalc", "resultat", somme);
    }

    ...
}
```

@RequestMapping indique que cette méthode répond à l'url « /simple/add », méthode GET.

Un exemple simple

```
@Controller
@RequestMapping("/simple")
public class ControleurSimple {

    @RequestMapping(value = "/add", method = RequestMethod.GET)
    public ModelAndView somme(int a, int b) {
        int somme = a + b;
        return new ModelAndView("simpleCalc", "resultat", somme);
    }

    ...
}
```

Les paramètre http sont automatiquement associés aux paramètres de la méthode. On attend donc deux paramètres, « a » et « b », de valeur entière

Un exemple simple

```
@Controller
@RequestMapping("/simple")
public class ControleurSimple {

    @RequestMapping(value = "/add", method = RequestMethod.GET)
    public ModelAndView somme(int a, int b) {
        int somme = a + b;
        return new ModelAndView("simpleCalc", "resultat", somme);
    }

    ...
}
```

Quand on retourne ModelAndView, on précise le *nom* de la vue.

Un exemple simple

```
@Controller
@RequestMapping("/simple")
public class ControleurSimple {

    @RequestMapping(value = "/add", method = RequestMethod.GET)
    public ModelAndView somme(int a, int b) {
        int somme = a + b;
        return new ModelAndView("simpleCalc", "resultat", somme);
    }

    ...
}
```

... et les données à placer dans le modèle. Ici, un bean nommé « resultat », dont la valeur est l'entier « somme ».
(on peut évidemment avoir plus d'un objet dans le modèle).

Un exemple simple

template simpleCalc.html

```
<!DOCTYPE html>
<html>
<head>
    <title>Résultat du calcul</title>
</head>
<body>
    Résultat du calcul :
    <span th:text="${resultat}">(00)</span>
</body>
</html>
```

Une template Thymeleaf. Automatiquement trouvée grâce à son nom, sous Springboot.

Un exemple simple

template simpleCalc.html

```
<!DOCTYPE html>
<html>
<head>
  <title>Résultat du calcul</title>
</head>
<body>
  Résultat du calcul :
  <span th:text="${resultat}">(00)</span>
</body>
</html>
```

L'attribut th:text va remplacer le corps de ce span par la valeur du bean resultat

Contrôleurs Spring

- Annotés avec @Controller
- Bean Springs (comme les @Components)
- trouvés automatiquement
- initialisés par Spring : injection de dépendances possible
- les méthodes du contrôleur seront associées à des URL...
- gère souvent un ensemble d'URL liées à un domaine (Façade web pour un *use case*, par exemple)
- remplace donc a priori plusieurs servlets...

@RequestMapping

(sur le Controller)

- Fixe un comportement général pour les méthodes gérées par le contrôleur
- Surtout utilisé pour fixer le début de l'URL :
`@RequestMapping("/simple")`

Méthodes d'un contrôleur

- Dans un contrôleur, une méthode annotée avec :
 - @RequestMapping
 - @PostMapping
 - @GetMapping
 - @PutMapping, @DeleteMapping, @PatchMapping
- gère une requête HTTP sur une URL donnée

@RequestMapping et ses attributs

- path ou value : url servie par la méthode ;
- method : méthode http pour laquelle la méthode (java) répond... à la place, souvent @PostMapping ou @GetMapping

Valeurs de retour

- Affichage en partie contrôlé par les valeurs de retour des méthodes du contrôleur ;
- **void** : méthode prend en charge de manière interne l'affichage - écrit sur OutputStream/Writer
- **@ResponseBody** : la valeur de retour contient les données à expédier - elle est éventuellement transformée
- **String** : nom d'une *vue* à afficher ;
- **ModelAndView** : couple nom de vue/ensemble de valeurs à communiquer à cette vue.

Arguments injectés

- Certaines valeurs peuvent être injectées directement comme argument:
 - HttpRequest, HttpResponse, HttpSession
 - Locale : pour l'internationalisation
 - OutputStream/Writer et InputStream/Reader : I/O bas niveau

Quelques exemples...


```

/**
 * Exemple d'affichage direct, sans passer par une vue.
 * @param out      : sortie pour affichage
 * @param response : la réponse (on en a besoin pour le type du contenu)
 */
@GetMapping(value = "/cercle")
public void cercle(OutputStream out, HttpServletResponse response)
    throws IOException {
    response.setContentType("image/png");
    int taille = 500;
    int marge = 10;
    int diametre = taille - 2 * marge;
    BufferedImage img =
        new BufferedImage(taille, taille, BufferedImage.TYPE_INT_RGB);
    Graphics g = img.getGraphics();
    g.setColor(Color.RED);
    g.fillOval(marge, marge, diametre, diametre);
    g.dispose();
    ImageIO.write(img, "png", out);
}

```

Injection de OutputStream

```

/**
 * Exemple d'affichage direct, sans passer par une vue.
 * @param out      : sortie pour affichage
 * @param response : la réponse (on en a besoin pour le type du contenu)
 */
@GetMapping(value = "/cercle")
public void cercle(OutputStream out, HttpServletResponse response)
    throws IOException {
    response.setContentType("image/png");
    int taille = 500;
    int marge = 10;
    int diametre = taille - 2 * marge;

```

On injecte le flux en sortie binaire (OutputStream) dans lequel on va écrire, ainsi que la réponse Http.

On pourrait se contenter de la réponse, qui a une méthode `getOutputStream()`

```

    g.setColor(COLOR.RED);
    g.fillOval(marge, marge, diametre, diametre);
    g.dispose();
    ImageIO.write(img, "png", out);
}

```

Injection de OutputStream

```

/**
 * Exemple d'affichage direct, sans passer par une vue.
 * @param out      : sortie pour affichage
 * @param response : la réponse (on en a besoin pour le type du contenu)
 */
@GetMapping(value = "/cercle")
public void cercle(OutputStream out, HttpServletResponse response)
    throws IOException {
    response.setContentType("image/png");
    int taille = 500;
    int marge = 10;
    int diametre = taille - 2 * marge;
    BufferedImage img =

```

On précise de quel type sera la réponse...

```

        g.dispose();
        ImageIO.write(img, "png", out);
    }

```

Injection de OutputStream

```

/**
 * Exemple d'affichage direct, sans passer par une vue.
 * @param out      : sortie pour affichage
 * @param response : la réponse (on en a besoin pour le type du contenu)
 */
@GetMapping(value = "/cercle")
public void cercle(OutputStream out, HttpServletResponse response)
    throws IOException {
    response.setContentType("image/png");
    int taille = 500;
    int marge = 10;
    int diametre = taille - 2 * marge;
    BufferedImage img =
        new BufferedImage(taille, taille, BufferedImage.TYPE_INT_RGB);
    Graphics g = img.getGraphics();
    g.setColor(Color.RED);
    g.fillOval(marge, marge, diametre, diametre);
    g.dispose();
    ImageIO.write(img, "png", out);
}

```

On écrit le résultat sur l'OutputStream.

Modèle et vue

- En Spring MVC, une méthode de contrôleur est supposée renvoyer :
 - le *nom* d'une *vue* qui servira à l'affichage
 - des données à placer dans cette vue - le « modèle »
- Pour cela :
 - renvoyer le nom de la vue (une String)
 - ou renvoyer un objet ModelAndView
 - on peut aussi recevoir un modèle utilisable comme argument

Exemple typique

```
@Controller
public class MessageController {

    @Autowired
    MessageRepository messageRepository;

    @GetMapping("/list2")
    public String getlist2(Model model) {
        model.addAttribute("messages",
                           messageRepository.getMessages());
        return "liste";
    }
}
```

Exemple typique

```
@Controller
public class MessageController {

    @Autowired
    MessageRepository messageRepository;

    @GetMapping("/list2")
    public String getlist2(Model model) {
        model.addAttribute("messages",
                           messageRepository.getMessage());
        return "liste";
    }
}
```



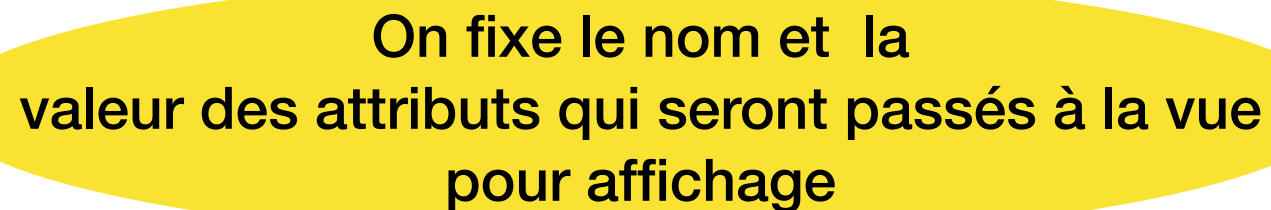
On retourne le nom de la vue

Exemple typique

```
@Controller
public class MessageController {

    @Autowired
    MessageRepository messageRepository;

    @GetMapping("/list2")
    public String getlist2(Model model) {
        model.addAttribute("messages",
                           messageRepository.getMessage());
        return "liste";
    }
}
```



On fixe le nom et la
valeur des attributs qui seront passés à la vue
pour affichage

Contenu de la template thymeleaf...

Du html...

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Liste de messages</title>
</head>
<body>
  <p th:each="m : ${messages}" th:text="${m}"></p>
</body>
</html>
```

des attributs qui visualisent les données du modèle...

Paramètres

Injection de paramètres simples

- Spring copie des paramètres http dans les arguments de méthodes
- conversion automatique

```
@RequestMapping(value = "/add", method = RequestMethod.GET)
public ModelAndView somme(int a, int b) {
    int somme = a + b;
    return new ModelAndView("simpleCalc", "resultat", somme);
}
```

ici, on attend deux paramètres, « a » et « b »

Injection de paramètres composites

- rappel « javabeau » = constructeur par défaut, getters et setters
- Si un argument du contrôleur est un bean java, ses propriétés sont remplies à partir des paramètres

```
@RequestMapping(value = "/add1", method = RequestMethod.GET)
public ModelAndView somme(SommeSimple forme) {
    int somme = forme.getA() + forme.getB();
    return new ModelAndView("simpleCalc", "resultat", somme);
}
```

```
public class SommeSimple {
    private int a,b;
    public SommeSimple() {}
    public int getA() {return this.a;}
    public void setA(int a) {this.a = a;}
    public int getB() {return this.b;}
    public void setB(int b) {this.b = b;}
}
```

Valeurs multiples (checkbox, select multiple)

- le problème : plusieurs champs avec le même « name »
- idéalement : une collection de valeurs ?

```
<form method="post">
  <input type="checkbox" name="choix" value="a"> a <br/>
  <input type="checkbox" name="choix" value="b"> b <br/>
  <input type="checkbox" name="choix" value="c"> c <br/>
  <input type="checkbox" name="choix" value="d"> d <br/>
  <input type="checkbox" name="choix" value="e"> e <br/>
  <input type="checkbox" name="choix" value="f"> f <br/>
  <input type="checkbox" name="choix" value="g"> g <br/>
  <input type="checkbox" name="choix" value="h"> h <br/>
  <input type="submit">
</form>
```

Valeurs multiples (checkbox, select multiple)

- Les paramètres sont correctement passés si le contrôleur a un argument qui **contient** une collection avec le bon nom, ou un argument qui est un **tableau** avec le bon nom
- ça ne fonctionne pas pour un argument qui serait une collection - probablement pour des questions d'effacement des paramètres formels.

Valeurs multiples (checkbox. select multiple)

```
<input type="checkbox" name="choix" value="a"> a <br/>  
<input type="checkbox" name="choix" value="b"> b <br/>
```

```
public class ChoixCheckBox {  
    private ArrayList<String> choix= new ArrayList<>();  
  
    public ArrayList<String> getChoix() {  
        return choix;  
    }  
  
    public void setChoix(ArrayList<String> choix) {  
        this.choix = choix;  
    }  
}
```

```
@PostMapping()  
public String post(ChoixCheckBox choixCheckBox, Model model) {  
    model.addAttribute("selection", choixCheckBox.getChoix());  
    return "listeChoix";  
}
```


@PathVariable

- Requête GET classique pour une page de forum : `http://forum/page?id=123`
- « Clean URL » : `http://forum/page/123`
- plus lisible, mieux indexé par google
- facile à réaliser dans Spring MVC avec @PathVariable

Bean sessions

- Composant Spring spécifique à chaque session d'un utilisateur
- annoté par @SessionScope
- le composant est détruit après un temps d'inactivité, ou après la fermeture de la session
- géré par un numéro de session conservé comme cookie
- *ça n'est pas directement un bean dans HttpSession*

Déclaration

```
@Component
@SessionScope
public class Compteur {

    private final AtomicInteger compteur= new AtomicInteger();

    public int getValeur() {
        return compteur.incrementAndGet();
    }

    public int incrementeValeur(int n) {
        return compteur.addAndGet(n);
    }
}
```

- attention : accès concurrents possibles (d'où utilisation de AtomicInteger ici)

Utilisation : par injection

```
@Controller
@RequestMapping("/sessions")
public class DemoSessionController {
    @Autowired Compteur compteur;

    @GetMapping
    public String index(Model model, HttpSession session) {
        model.addAttribute("valeurCompteur", compteur.getValeur());

        return "sessions/index";
    }
}
```

Problème...

- Mais mais mais...
- le Compteur est *a priori* partagé par tous les utilisateurs, tel qu'il apparaît ici...
- ça n'est pas du tout ce qu'on veut...
- pourtant... ça fonctionne !
- comment ?

Un peu de magie Spring

- En fait, l'objet injecté dans le contrôleur n'est pas l'objet compteur lui-même, mais un *proxy* généré dynamiquement
- Ce proxy renvoie pour chaque accès à l'instance de Compteur créé pour la session en cours.

Autres objets en session

- Pour un objet conservé directement dans HttpSession :
 - on peut accéder directement à HttpSession et utiliser les méthodes « normales » des servlets
 - on peut, si l'objet existe, le récupérer comme argument d'un contrôleur - il faut annoter l'argument avec @SessionAttribute (sans s !!!)

@SessionAttributes

- (le « s » est important)
- Pour des *attributs temporaires* (liés à des « conversations »)
- marque une classe et donne la liste des paramètres de méthodes à stocker en session
- Méthode setComplete() de SessionStatus : oublie les attributs de session ***du contrôleur courant***.

@SessionAttributes

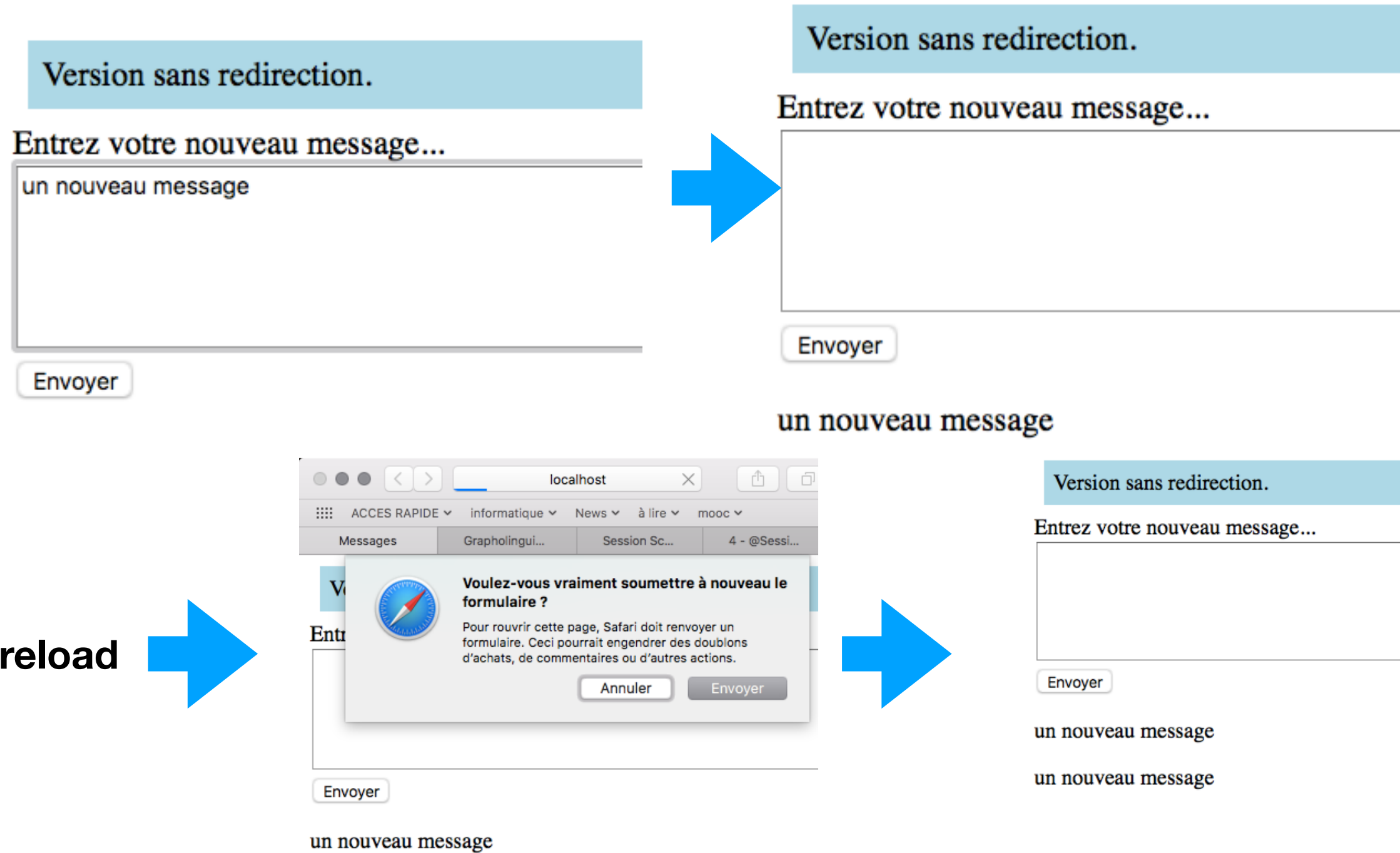
```
@Controller
@SessionAttributes({"dateAttribute"})
public class DemoSessionAttributeController {

    @GetMapping("/sessions/attr")
    @ResponseBody
    public String getDateSession(DateAttribute dateAttribute) {
        return dateAttribute.toString();
    }

    @GetMapping("/sessions/attr/clear")
    @ResponseBody
    public String clear(SessionStatus status) {
        status.setComplete();
        return "session terminée";
    }
}
```

Redirection

- Le problème...



Explication

- Quand on recharge la page, on rejoue le « POST » ;
- on renvoie donc une seconde fois le formulaire
- ... et le message est envoyé en double.

Solution:

- Utiliser une redirection pour l'affichage
- Une redirection demande au client de charger une nouvelle page, en mode GET...
- et le POST n'est pas rejoué

```
@GetMapping("/form")
public String getForm2(Model model) {
    return "formulaireEtListe";
}
```

```
@PostMapping("/form")
public String postForm2(String nouveauMessage, Model model) {
    if (Strings.isEmpty(nouveauMessage)) {
        model.addAttribute("erreur", "message vide");
    } else {
        messageRepository.add(nouveauMessage);
        return "redirect:/list";
    }
}
```

renvoie vers l'URL « /list » en mode GET

Moteurs de templates :

Thymeleaf

- But : décrire une page à afficher, avec un contenu variable
- la template ne calcule pas de valeurs, elle décide uniquement comment afficher celles que lui fournit le modèle
- une page *thymeleaf* est une page html « presque » normale. Affichable sans interprétation
- Thymeleaf étend le langage html avec de nouveaux attributs (commençant par « th: »)

Le principal sur Thymeleaf

- compatibilité avec HTML
- le langage Spring EL, les bases
- affichage de valeurs
- boucles
- conditionnelles

Visualisation "brute"

- Le HTML d'une page thymeleaf est visualisable directement → pratique pour la création

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Message</title>
</head>
<body>
  <h1>Affichage d'un message</h1>

  <p th:text="${message.texte}">
    Lorem ipsum dolor sit amet,
    consectetur adipiscing elit. Sed non risus.
    Suspendisse lectus tortor,
    dignissim sit amet, adipiscing nec, ultricies sed, dolor
    .
    Cras elementum ultrices diam. .</p>

  <address>Auteur:
    <span th:text="${message.auteur}">Cicero</span></address>
</body>
</html>
```



Affichage d'un message

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed non risus. Suspendisse lectus tortor, dignissim sit amet, adipiscing nec, ultricies sed, dolor. Cras elementum ultrices diam. Maecenas ligula massa, varius a, semper congue, euismod non, mi. Proin porttitor, orci nec nonummy molestie, enim est eleifend mi, non fermentum diam nisl sit amet erat. Duis semper. Duis arcu massa, scelerisque vitae, consequat in, pretium a, enim. Pellentesque congue.

Auteur: Cicero

Fixer le contenu d'un élément html : th:text

- attribut **th:text**
- remplace le contenu d'une balise par la valeur de l'attribut
- les valeurs des balises utilisent le *Spring Expression Language*

```
<address>Auteur:  
    <span th:text="${message.auteur}">Cicero</span>  
</address>
```



Le contenu de l'élément (« Cicero ») sera remplacé par la valeur de message.auteur. message est un élément du modèle (au sens ModelAndView)

protection contre l'injection javascript

- **th:text** protège automatiquement le texte qu'on lui fournit

Entrez votre nouveau message...

Un utilisateur malveillant saisit du code
<script>alert("attaque")</script> agressif|

Envoyer

à l'affichage, le html sera :

Un utilisateur malveillant saisit du code
<script>alert("attaque")</script> agressif

Raccourci

- Pour l’affichage direct de texte, on dispose aussi de `[[...]]` :

`<address>Auteur: [[${message.auteur}]]</address>`

Variante sans protection

- Quand on veut explicitement injecter du HTML voire du javascript
- `th:utext`
- `[(...)]`

Spring EL

- Dans Spring EL, la notation `${...}` permet
 - d'accéder à des objets du modèle : `${total}`
 - ... à leurs propriétés : `${utilisateur.nom}` (**appelle `getNom()` !**)
 - accès à des entrées de maps : `${map.nomClef}`
 - appels de procédures, etc...

URL : @{...}

- Sur un serveur, l'URL commence a priori par le nom de l'application web
- @{...} affiche une URL en y ajoutant automatiquement le chemin de l'application web
- permet d'injecter des valeurs de paramètres, etc...

```
'http://localhost:8080/gtvg/order/details?orderId=3'  
@{http://localhost:8080/gtvg/order/details(orderId=${o.id})}
```

```
'/gtvg/order/details?orderId=3'  
@{/order/details(orderId=${o.id})}
```

```
<!-- '/gtvg/order/3/details' -->  
@{/order/{orderId}/details(orderId=${o.id})}
```

valeurs d'attributs HTML

- Pour la plupart des attributs : th:nomAttribut :

```
<a href="#" th:href="@{/redirection/form2}">Retour</a>
```

- La valeur fournie « en dur » (ici '#') ne sera pas utilisée - mais est utile pour visualiser la page sans filtrage par thymeleaf

- Cas général : th:attr

```
<a href="#" th:attr="href=@{/redirection/form2}">Retour</a>
```

Boucle

- Pour les tableaux, listes, etc...
- parcourt typiquement une collection placée dans le modèle:

```
<ul>  
  <li th:each="c : ${selection}" th:text="${c}">un choix...</li>  
</ul>
```

c va parcourir la collection « selection »

on utilise c pour l'affichage

Boucle

- Pour les tableaux, listes, etc...
- parcourt typiquement une collection placée dans le modèle:

```
<ul>  
  <li th:each="c : ${selection}" th:text="${c}">un choix...</li>  
</ul>
```

- généralement, on accède aux propriétés des éléments :

```
<p th:each="m : ${messages}">  
  <span th:text="${m.texte}">(texte)</span> par  
  <em th:text="« ${m.auteur} »>..auteur...</em>  
</p>
```

Conditionnelles : th:if

- N'affiche une balise que si la condition est vraie
- th:if est souple : il teste les booléens, les collections vides, etc.

```
<div class="erreur" th:if="{erreur}" th:text="{erreur}"></div>
```

Pseudo-balise th:block

- le système de thymeleaf nous conduit à introduire des balises ou <div> artificielles

```
<p th:each="m : ${messages}">  
  <span th:text="${m.texte}">(texte)</span> par  
  <span th:text="${m.auteur}">..auteur...</span>  
</p>
```



```
<p> <span>texte du message 1</span> par  
    <span>auteur du message 1</span></p>  
<p> <span>texte du message 2</span> par  
    <span>auteur du message 2</span></p>
```

Pseudo-balise th:block

- th:block permet d'introduire du contenu Spring sans appuyer sur une balise.

```
<p th:each="m : ${messages}">  
  <th:block th:text="${m.texte}">(texte)</th:block> par  
  <th:block th:text="${m.auteur}">..auteur...</th:block></em>  
</p>
```



```
<p> texte du message 1 par auteur du message 1</p>  
<p> texte du message 2 par auteur du message 2</p>
```

Formulaires et validation

- Spring MVC facilite la validation des formulaires
- il effectue automatiquement les conversions
- un contrôleur peut recevoir un objet BindingResult pour stocker les erreurs
- Attention !! l'argument BindingResult **doit immédiatement suivre le modèle de formulaire dans la liste des arguments.**
- le modèle du formulaire peut être annoté en suivant la JSR 303
- Thymeleaf simplifie l'affichage du formulaire et des messages d'erreur.

Saisie et Ajout d'un produit avec Validation

ProduitController.java

```
@RequestMapping(value="/admin/form",method=RequestMethod.GET)
public String form(Model model){
    model.addAttribute("produit", new Produit());
    return "FormProduit";
}
@RequestMapping(value="/admin/save",method=RequestMethod.POST)
public String save(Model model,@Valid Produit p,BindingResult bindingResult){
    if(bindingResult.hasErrors()){ return "FormProduit"; }
    produitRepository.save(p);
    return "Confirmation";
}
```

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

@Entity

```
public class Produit implements Serializable {
    @Id @GeneratedValue
    private Long id;
    @NotNull
    @Size(min=4,max=70)
    private String designation;
    @DecimalMin(value="100")
    private double prix;
    private int quantite;
}
```

Produit.java

Validation

```
<form th:action="@{/admin/save}" method="post">
    <label class="control-label">Désignation:</label>
    <input type="text" name="designation" th:value="${produit.designation}"/>
    <span th:errors="${produit.designation}" ></span>

    <label>Prix:</label>
    <input type="text" name="prix" th:value="${produit.prix}"/>
    <span th:errors="${produit.prix}" ></span>

    <label>Quantité:</label>
    <input type="text" name="quantite" th:value="${produit.quantite}"/>
    <span th:errors="${produit.quantite}" ></span>

    <button type="submit">Save</button>
</form>
```

FormProduit.html

Exemple simple

```
<form action="#" th:action="@{/calc}" th:object="${calcForm}"
      method="post">
  <input type="text" th:field="*{a}"/>
  <input type="text" th:field="*{b}"/>
  <input id="submit" type="submit"/>
</form>
```

- th:action : (optionnel) : renvoie vers l'action du formulaire
- th:object : stocke les données du formulaire dans cet objet
- *{a} : propriété « a » de l'objet calcForm
- gère les names, les ids, le réaffichage en cas d'erreur

Architecture conseillée pour le contrôle

- Une méthode get et une méthode post pour la même URL
- get : affichage de la page
- post : traitement du formulaire


```

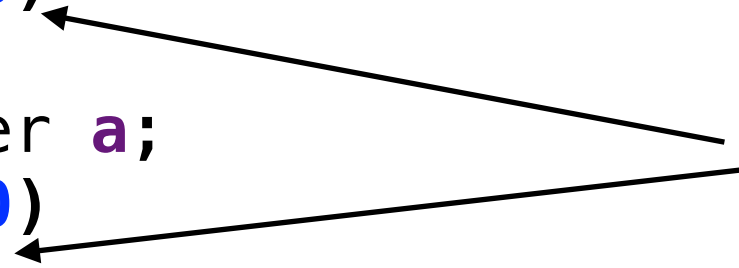
@GetMapping
public String getForm(CalcForm calcForm) {
    return "calcForm";
}

@PostMapping
public ModelAndView processForm(
    @Valid CalcForm calcForm, // Formulaire validé
    BindingResult bindingResult // erreurs
) {
    if (bindingResult.hasErrors()) {
        // si erreur: on réaffiche le formulaire
        return new ModelAndView("calcForm");
    } else {
        // sinon: on calcule et affiche le résultat
        return new ModelAndView("calcForm", "resultat", calcForm.sum());
    }
}

```

Modèle du formulaire

```
public class CalcForm {  
    @Min(value = 0)  
    @NotNull  
    private Integer a;  
    @Min(value = 0)  
    @NotNull  
    private Integer b;  
  
    public Integer getA() {return a;}  
    public void setA(Integer a) {this.a = a;}  
    public Integer getB() {return b;}  
    public void setB(Integer b) {this.b = b;}  
}
```



annotations JSR 330

Validation

- On valide avec les annotations JSR 330
- Elles peuvent éventuellement comporter un message personnalisé comme attribut
- La validation est paramétrable : on peut créer ses propres validateurs (Validator)

Formulaire avec erreurs

```
<form action="#" th:object="${calcForm}" method="post">
  <input type="text" th:field="*{a}" th:errorclass="fieldError"/>
  <p th:if="${#fields.hasErrors('a')}" th:errors="*{a}">err</p>
  <input type="text" th:field="*{b}" th:errorclass="fieldError"/>
  <p th:if="${#fields.hasErrors('b')}" th:errors="*{b}"></p>
  <input type="submit"/>
</form>
```

```
<input type="text" th:field="*{a}" th:errorclass="fieldError"/>
```

- `th:errorclass`: active la classe en question s'il y a une erreur sur le champ. En cas de problème, on aura : `class='fieldError'`.

Formulaire avec erreurs

```
<form action="#" th:object="${calcForm}" method="post">
  <input type="text" th:field="*{a}" th:errorclass="fieldError"/>
  <p th:if="${#fields.hasErrors('a')}" th:errors="*{a}">err</p>
  <input type="text" th:field="*{b}" th:errorclass="fieldError"/>
  <p th:if="${#fields.hasErrors('b')}" th:errors="*{b}"></p>
  <input type="submit"/>
</form>
```

```
<p th:if="${#fields.hasErrors('a')}" th:errors="*{a}">err</p>
```

si le champ comporte des erreurs (#field = objet auxiliaire avec méthodes pratiques pour manipuler les champs), on les affiche avec th:errors (qui fonctionne comme th:text, en remplaçant le contenu d'un élément).

Autres fonctionnalités

- Templates : permet de construire des bouts de page Thymeleaf
- Internationalisation facile (traduction des textes fixes d'une application)
- Négociation de contenu : capacité de renvoyer des formats de données différents.
- facilités pour REST (cours à venir)

Bibliographie

- <https://docs.spring.io/spring/docs/current/spring-framework-reference/pdf/web.pdf>
- <https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.pdf>
- <https://www.thymeleaf.org/doc/tutorials/3.0/thymeleafspring.pdf>
- Configuration de Thymeleaf sans Spring boot: <https://www.thymeleaf.org/doc/tutorials/3.0/thymeleafspring.html>
-