



# Spring MVC

En mode JSP / Servlet

Présentation

# Spring MVC JSP / Servlet

2

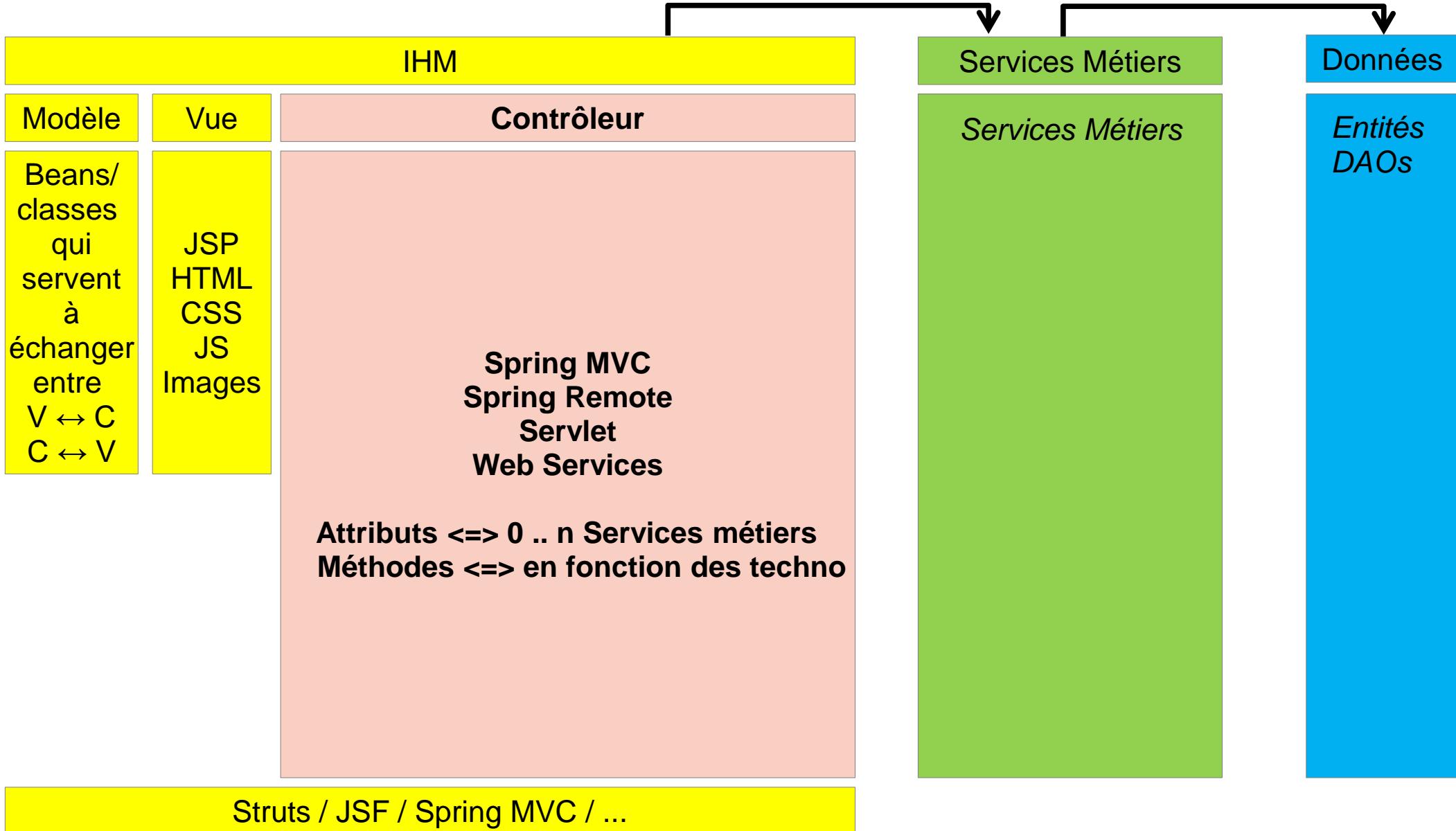
- ✓ Présentation
- ✓ Objets principaux
- ✓ Annotations Spring MVC (@Controller)
- ✓ Gestion des erreurs
- ✓ Gestion des validations de bean
- ✓ Internationnalisation

# Présentation

- Spring dispose d'un framework web intégrant :
  - ▶ Le Pattern MVC2
  - ▶ Une technologie de binding et de validation
  - ▶ Une très grande souplesse pour la technologie de présentation utilisée (html, wml, PDF, Excel...)
- Ce chapitre ne présente que la partie JSP/Servlet
  - ▶ Ne contient pas la partie sur les services REST

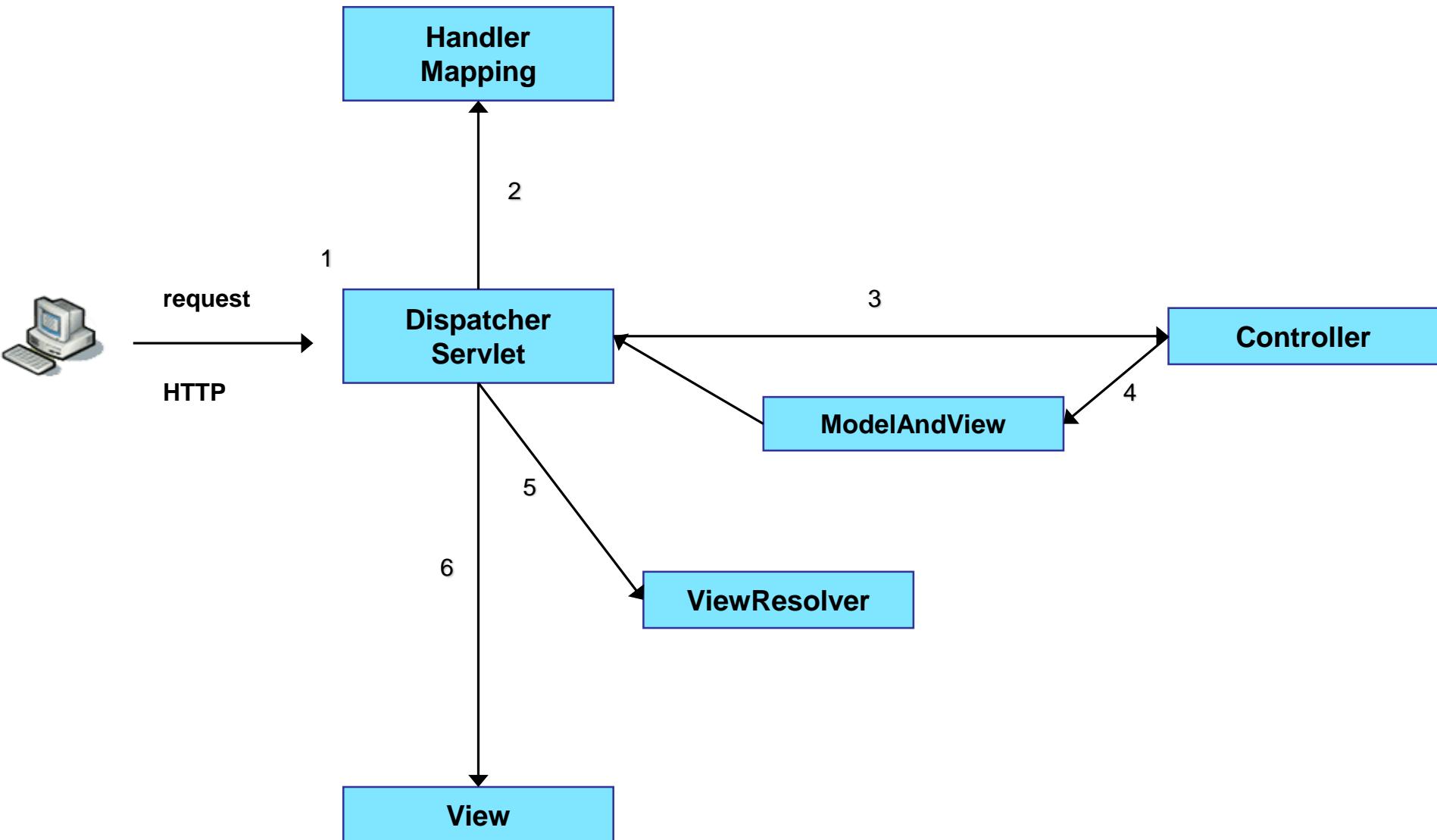
# Présentation

4



# Cycle de vie d'une requête

5



# Config Spring dans une application Web

- La configuration s'effectue dans le fichier web.xml
  - ▶ Vous devez indiquer où sont vos fichiers Spring en ajoutant un context-param dans votre fichier web.xml

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath*:spring/*-context.xml</param-value>
</context-param>
```

- On parle ici des fichiers propres à la couche métier (@Repository, @Service)
  - ▶ Vous devez aussi déclarer un listener qui va s'occuper de charger le/les fichiers spécifiés

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

# Config Spring dans une application Web

7

- En mode web, vous ne devez **surtout pas** créer un ClassPathXmlApplicationContext pour le charger.
- Si vous avez besoin d'un bean dans vos contrôleurs, utilisez @Autowired
  - ➡ N'oubliez pas de déclarer le bean et de préciser le component-scan adéquate

# Config Spring dans une application Web

- La configuration du Spring MVC se complètera avec (toujours dans le fichier web.xml)
  - La déclaration de la servlet principale du Spring

□ Exemple :

```
<servlet>
  <servlet-name>spring</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
```

□ Rappel : load-on-startup force conteneur J2EE à charger la servlet au démarrage du serveur (et pas lors de sa première utilisation)

# Config Spring dans une application Web

- Le DispatcherServlet aura besoin d'un fichier de configuration Spring dédié
- Par défaut, utilisation de `/WEB-INF/${nom-servlet}-servlet.xml`
  - ➡ Ici, le fichier utilisé sera `/WEB-INF/spring-servlet.xml`
- On parle ici du fichier de configuration des `@Controller`

# Config Spring dans une application Web

- Vous pouvez surcharger la localisation du fichier de configuration
  - Via un paramètre d'initialisation (init-param)

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

# Config Spring dans une application Web

- Après la déclaration du DispatcherServlet, il faut spécifier son mapping
  - Les urls utilisées doivent suivre un mapping commun
  - Tous les mappings sont possibles.
    - Ex : \*.smvc
  - Nous sommes toujours dans le fichier web.xml

```
<servlet-mapping>
<servlet-name>spring</servlet-name>
<url-pattern>*.smvc</url-pattern>
</servlet-mapping>
```

# Framework Spring MVC

12

- Il existe plusieurs manières de faire du Spring MVC
  - ➡ A l'ancienne ⇔ Struts 1 : héritage, méthode figée
    - ❑ On sait quoi écrire, on est comme dans une servlet
    - ❑ Simple, sans prise de tête si on vient du MVC standard JSP/Servlet (2000/2004)
    - ❑ On fait usage de HttpServletRequest, HttpServletResponse
      - Compliqué pour les tests unitaires ⇔ obligé de faire des tests d'intégrations sur la partie des contrôleurs
  - ➡ Moderne : annotation, méthodes dynamiques
    - ❑ On peut faire ce que l'on veut, paramètre, nom de méthodes, ...
    - ❑ On n'est plus obligé de faire usage de HttpServletResponse , HttpSession, HttpServletRequest ...
      - On peut faire des tests unitaire sur nos contrôleur
    - ❑ Peut devenir complexe à gérer dans le temps, car chacun peut faire à sa sauce

# Framework Spring MVC – Ancienne

13

- Vos contrôleurs
  - ▶ Implémente l'interface Controller
  - ▶ Le contrôleur le plus simple proche de la servlet
  - ▶ A vous de faire les request.getParameter ...
- Ou, si vous avez besoin d'un peu plus de service, vos contrôleurs peuvent hériter de AbstractCommandController
  - ▶ Ce type de contrôleur permet un binding automatique des paramètres d'une requête vers un objet
    - Ex : sélection d'un élément dans une liste
  - ▶ Attention, un CommandController n'est pas lié à un formulaire

# Framework Spring MVC – Ancienne

14

- Vos formulaires :
  - ▶ Hérite de SimpleFormController
  - ▶ Gère les opérations liées à un formulaire
    - Population automatique des éléments saisis
    - Invocation d'une validation
- Ou peuvent faire usage d'un Wizard
  - ▶ Hérite de AbstractWizardFormController
  - ▶ Permet de gérer un formulaire saisi sur plusieurs pages
    - Un contrôleur peut être associé à plusieurs vues
    - Possibilité de paramétriser une méthode de validation par page

# Framework Spring MVC – Ancienne

15

- Création de la classe contrôleur
- Configuration du contrôleur dans le contexte Spring
- Création de la classe Form associé au contrôleur si besoin
- Configuration du view resolver (mapping controller/jsp)
- Création de la page JSP et utilisation des tag lib Spring

# Framework Spring MVC – Ancienne

## ➤ La Classe contrôleur

- ▶ Implémente l'interface Controller
- ▶ La méthode handleRequest renvoie un objet ModelAndView
  - Nom de la vue + données d'affichage

```
public class SimpleController implements Controller {  
  
    public ModelAndView handleRequest(HttpServletRequest request,  
                                         HttpServletResponse response) throws Exception {  
  
        Map map = new HashMap();  
        map.put("message", "ceci est un message a afficher");  
        map.put("id", "identification");  
        return new ModelAndView("jsp/main.jsp", map);  
    }  
}
```



En Spring 3+, il faut utiliser les annotations.

# Framework Spring MVC – Ancienne

17

- Il vous faudra mapper chaque contrôleurs dans le fichier Spring du DispatcherServlet

- ➡ Exemple de syntaxe pour le mapping

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

<bean name="/simple.smvc" class="com.bankonet.SimpleController"/>
```



Contrairement aux utilisations précédentes, c'est l'attribut "name" qui est utilisé (et non-pas "id")

- ➡ Le suffixe n'est pas contractuel.

- Il faut juste qu'il soit cohérent avec le suffixe dans le web.xml

- ➡ Il est possible d'ajouter des propriétés à ce bean

- Ex : injection de dépendance avec une classe Service

# Spring MVC et les annotations

- Spring 2.5 a introduit des annotations pour définir les contrôleurs
- Idées :
  - ➡ Supprimer la nécessité d'héritage
  - ➡ Privilégier le modèle POJO (annoter des classes POJO)
  - ➡ Alléger la configuration XML
  - ❑ Plus besoin du mapping

# Framework Spring MVC – Moderne

19

## ➤ Le contrôleur (en annotations)

- ▶ Pas d'héritage de classe/interface spécifique
- ▶ Pas de méthode spécifique

```
@Controller
@RequestMapping(value = "/simple.smvc")
public class SimpleController {

    @RequestMapping(method = RequestMethod.GET)
    public String faireQQchose(HttpServletRequest request, HttpServletResponse response) throws Exception {
        Map map = new HashMap();
        map.put("message", "ceci est un message a afficher");
        map.put("id", "identification");
        request.setAttribute("map", map);
        return "jsp/main.jsp";
    }

    @RequestMapping(method = RequestMethod.POST)
    public String faireAutreChose(HttpServletRequest request, HttpServletResponse response) throws Exception {
        ...
        return "jsp/main.jsp";
    }
}
```

# Le view resolver

- Cet élément à pour objectif de simplifier vos chemins d'URL dans le code
- Déclaration du ViewResolver

```
<bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
<property name="viewClass" value="org.springframework.web.servlet.view.JstlView" />
<property name="prefix" value="/jsp/" />
<property name="suffix" value=".jsp" />
</bean>
```
- La propriété viewClass peut prendre d'autre valeur en fonction des framework (JSF par exemple) ou contraintes
- Exemple de transformation
  - ▶ Dans le contrôleur :

```
Map map = new HashMap();
map.put(...);
return new ModelAndView("home",map);
```
  - ▶ home ⇔ /jsp/home.jsp

# ModelAndView

- Contient :
  - La vue (identifiant de la vue)
  - Le modèle (ensemble des informations à afficher)
- Les attributs du constructeur sont inversés !!
  - La vue est le premier argument
- Spring-MVC place automatiquement la Map dans le contexte approprié (request par défaut)

```
Map map = new HashMap();
map.put(...);
return new ModelAndView("home", map);
```

# Contrôleur exemple (1/4)

- Déclaration, instantiation et injection de dépendances
- Rappel : un contrôleur **n'est pas** un service métier

```
@Controller  
@RequestMapping("/login.smvc")  
public class LoginController {  
  
    private final IClientService clientService;  
  
    @Autowired  
    public LoginController(IClientService clientService) {  
        this.clientService = clientService;  
    }  
  
    ...  
}
```

Exemple :  
Injection via le  
constructeur

# Contrôleur exemple (2/4)

23

## ➤ Affichage du formulaire

```
@Controller  
@RequestMapping("/login.smvc")  
public class LoginController {  
  
    ...  
  
    @RequestMapping(method=RequestMethod.GET)  
    public String setupForm(ModelMap model) {  
        model.addAttribute("client", new Client());  
        return "clientForm";  
    }  
}
```

# Contrôleur exemple (3/4)

24

## ➤ Soumission du formulaire

```
@Controller  
@RequestMapping("/login.smvc")  
public class LoginController {  
  
    ...  
  
    @RequestMapping(method=RequestMethod.POST)  
    public String processSubmit(@ModelAttribute("client") Client client,  
                                BindingResult error, HttpServletRequest request, ModelMap map) {  
  
        client = clientService.findClientByNom(client.getNom());  
  
        if(client == null) {  
            error.reject(null, "Impossible de trouver le client !");  
            return "clientForm";  
        }  
  
        map.addAttribute("client", client);  
        return "clientDetail";  
    }  
}
```

# Déclaration (4/4)

## ➤ Fichier de contexte Spring du DispatcherServlet

```
<mvc:annotation-driven />

<context:component-scan base-package="com.banque.web" />

<bean id="jspViewResolver"
      class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="viewClass" value="org.springframework.web.servlet.view.JstlView" />
    <property name="prefix" value="/WEB-INF/jsp/" />
    <property name="suffix" value=".jsp" />
</bean>
```

- N'oubliez pas le component-scan sur vos classes @Controller
  - Sinon pas de @Autowired avec vos services métiers

# Annotations Spring MVC, conclusion

26

- Les vues n'ont pas changé
- Approche 100% POJO
  - ▶ Plus à se préoccuper de quelle classe de contrôleur hériter
- Aucune configuration XML pour les contrôleurs
  - ▶ Vérification à la compilation
- Approche décentralisée
  - ▶ Avantage : pas de fichier de configuration à maintenir
  - ▶ Inconvénient : difficile d'avoir une vue globale
- Moins de réutilisabilité
- Nécessite des conventions de nommage pour des projets à grande échelle

# Définition Spring MVC

27

- La signature des méthodes n'est pas figée.

- En paramètres :

Request, Response et Session de l'API Servlet	
@PathVariable	récupère une partie de l'url et la place dans le paramètre annoté.
@RequestParam	récupère un paramètre de requête web (GET ou POST)
@RequestHeader	récupère un header de requête web
@RequestBody	accède au corps de la requête
@ModelAttribute	classe contenant le modèle objet et permet son enrichissement avant transmission à la vue

- En retour :

ModelAndView	Encapsule le nom logique de la vue et son modèle.
Model	Encapsule le modèle. Retour sur la vue courante.
View	Retourne la vue préparée par l'application. Exemple : retour de documents générés.
String	Fournit le nom logique

# Spring MVC : @RequestMapping

- Annotation de classe et/ou de méthode
- Permet d'indiquer l'URL de votre web service
- Sur une classe : Indique la racine de l'URL
  - `@RequestMapping("/comptes")`
- Sur une méthode : s'additionne au chemin de la classe
  - `@RequestMapping(value = "/lister" ...)`
  - `/comptes/lister`

# Spring MVC - @RequestMapping

29

- Exemple de déclaration

```
1 package com.banque.web.controller;  
2  
3+ import java.util.List;  
17  
19+ * Controller qui liste les comptes <br/>..  
21 @Controller  
22 @RequestMapping(value = "/listeCompte.smvc")  
23 @SessionAttributes({ "utilisateur" })  
24 public class CompteController {
```

- Ce contrôleur sera joignable sur l'URL

- [http\(s\)://\[server\]:\[port\]/\[contextjee\]/\[urlmappingspringmvc\]/\[requestmappingclass\]](http://[server]:[port]/[contextjee]/[urlmappingspringmvc]/[requestmappingclass])
  - <http://localhost:8080/netbank/listeCompte.smvc>

- Ce contrôleur sera lié par injection de dépendance avec notre service métier de gestion des comptes

# Spring MVC : @RequestMapping

- @RequestMapping
  - ▶ s'additionne au chemin de la classe
    - @RequestMapping(value = "/lister" ...)
    - URL = classe + méthode ⇔ /comptes/lister
  - ▶ permet de définir
    - le type de données en **entrées** et en **sorties**
    - les commandes HTTP supportées (get, put, ...)
      - @RequestMapping(method = { RequestMethod.GET, RequestMethod.PUT })

# Spring MVC - @RequestMapping

31

## ➤ Exemple :

```
67*   * Affiche la page des virements.□
68
69
70
71
72
73
74
75  @RequestMapping(value = "/virement.smvc", method = RequestMethod.GET)
76  public String showVirement(Model model, ModelMap modelMap) {
77      VirementController.LOG.debug("--> Passage dans showVirement");
78      VirementBean virementBean = new VirementBean();
79      model.addAttribute("virementBean", virementBean);
80      // c'est l'annotation qui fait le lien avec la session
81      IUtilisateurEntity utilisateur = (IUtilisateurEntity) modelMap.get("utilisateur");
82      if (utilisateur == null) {
83          VirementController.LOG.error("Erreur : utilisateur non connecté");
84          return "index";
85      }
86      this.getAndSetListeComptes(model, utilisateur.getId());
87      return "comptes/virement";
88  }
```

► Le web service sera joignable sur l'URL

- `http(s)://[server]:[port]/[contextjee]/[urlmappingspringmvc]/[requestmappingclass]/[requestmappingmethode]`
- `http://localhost:8080/netbank/vr/virement.smvc`

► Il fonctionnera en get uniquement

# Spring MVC : @RequestParam

32

- Annotation sur les paramètres de la méthode
  - ... faireQQC(@RequestParam String login, ...)
- Permet de récupérer un paramètre provenant d'un formulaire ou d'un URL
- Le typage du paramètre n'a pas obligation d'être une chaîne de caractères
- Vous devez indiquer le name du paramètre dans l'annotation
  - C'est le name dans le formulaire ou la clef dans votre URL
  - Remarque : en spring < 4, il n'y a pas d'attribut name pour cette annotation, le nom du paramètre Java DOIT être le name du parameter.
- Vous pouvez ajouter l'option *required* dans l'annotation pour indiquer le fait que le paramètre est obligatoire

# Spring MVC : @RequestParam

33

- C'est Spring qui s'occupera de gérer la conversion si le typage n'est pas une String
- Si vous indiquez comme typage un objet à vous, et que le consumes indique JSON ou Xml, c'est aussi le Spring qui se chargera de faire la transformation
  - ❑ JAX-B pour l'XML
  - ❑ Jackson pour le Json où nom attribut = clef json
    - Attention aux majuscules / minuscules
- C'est aussi Spring qui va gérer les erreurs :
  - ❑ code 400 si le paramètre est absent (et en *required=true*)

# Spring MVC - Définition d'une méthode

- Exemple de méthodes :
  - Récupération automatique d'un paramètre

```
//Méthode appelée par défaut pour toute requête GET http.  
@RequestMapping(method = RequestMethod.GET)  
public String methode (@RequestParam("id") int id, Model model) {  
    ...  
}
```

- Récupération d'un header

```
@RequestMapping("/info")  
public void Info(@RequestHeader("Accept-Encoding") String encoding,  
                 @RequestHeader("Keep-Alive") long keepAlive) {  
    ...  
}
```

- En théorie vous n'avez plus besoin de  
HttpServletRequest

# Gestion des vues

- Chaque méthode de contrôleur est chargée de sélectionner la vue consécutive à son traitement. Pour cela, elle renvoie
  - ▶ un nom logique en fin de traitement.

```
// Ou d'une autre manière si aucun modèle.
```

```
@RequestMapping("/helloWorld")
public String helloWorld() {
    return "helloWorld";
}
```

- ▶ Ou un objet ModelAndView

```
@RequestMapping("/helloWorld2")
public ModelAndView helloWorld() {
    ModelAndView mav = new ModelAndView();
    mav.setViewName("helloWorld");
    mav.addObject("message", "Hello World!");
    return mav;
}
```

# Gestion des vues

36

- Dans l'exemple précédent, la méthode `helloWorld` fournit à la vue les données par le biais du modèle transmis.
- La classe `ModelAndView` gère non seulement le nom logique de la vue mais aussi le modèle de la vue.
- Depuis la vue, l'attribut du modèle est directement référençable comme tout bean exposé à une page web.

# Gestion du modèle

37

- Quand vous voulez faire passer un bean représentant le modèle, vous ferez usage de l'annotation `@ModelAttribute` sur un des paramètres de votre méthode
  - Le typage sera celui du bean qui vous intéresse

```
88+     * Soumet le formulaire.□
98@    @RequestMapping(value = "/dovirement.smvc", method = RequestMethod.POST)
99     public String doVirement(@ModelAttribute("virementBean") VirementBean virementBean, ModelMap modelMap,
100                           BindingResult bindingResult) {
101     VirementController.LOG.debug("--> Passage dans doVirement");
```

# Gestion du modèle

38

- **ModelAndView** : vue + modèle : on place dedans des informations en relation avec la vue.
  - ▶ Utiliser comme retour de méthode.
- **ModelMap** ou **Model** : Model si vous êtes en Java5 sinon ModelMap. On place dedans des informations, ce que vous voulez. C'est un peu le xxxx.setAttribute (ou xxx est un scope). Par défaut, tout va dans le scope request.
  - ▶ Utiliser comme paramètre de méthode.
- **@ModelAttribute** : permet de récupérer un bean qui sera automatiquement construit par Spring
  - ▶ Utiliser sur un paramètre de méthode ou une méthode pour cibler le code retour.
  - ▶ Fait le lien entre un formulaire web et le code

# Gestion du modèle – scope session

39

- Pour rappel, vous pouvez ajouter un paramètre HttpServletRequest comme paramètre de vos méthodes
  - ▶ A partir de là, vous avez accès à tous les scopes
  - ▶ Mais c'est maladroit
- La gestion du scope des éléments que vous placez dans le Model / ModelAndView / ModelMap se fera via l'annotation **@SessionAttributes**
  - ▶ C'est une annotation de classe
- Attention : l'annotation de classe @Scope **n'a pas de rapport** avec notre J2EE mais avec le scope du contexte Spring !

# Gestion du modèle – scope session

40

## ➤ Exemple

```
27 @Controller  
28 @SessionAttributes( { "utilisateur" } )  
29 public class VirementController {
```

```
66+     * Affiche la page des virements.  
72@     @RequestMapping(value = "/virement.smvc", method = RequestMethod.GET)  
73     public String showVirement(ModelMap modelMap) {  
74         VirementController.LOG.debug("--> Passage dans showVirement");  
75         VirementBean virementBean = new VirementBean();  
76         modelMap.addAttribute("virementBean", virementBean);  
77         // c'est l'annotation qui fait le lien avec la session  
78         IUtilisateurEntity utilisateur = (IUtilisateurEntity) modelMap.get("utilisateur");  
79         if (utilisateur == null) {  
80             VirementController.LOG.error("Erreur : utilisateur non connecté");  
81             return "index";  
82         }  
83         this.getAndSetListeComptes(modelMap, utilisateur.getId());  
84         return "comptes/virement";  
85     }
```

# Gestion du modèle – scope application

- Pour accéder au scope de l'application, vous devez ajouter un objet de type javax.servlet.ServletContext dans votre contrôleur
- Et lui indiquer qu'il est en **@Autowired**
  - ➡ Ou, ne pas mettre @Autowired et implémenter l'interface org.springframework.web.context.ServletContextAware
- Votre attribut sera géré par le Spring et vous donnera accès au scope application

# Gestion des JSP

42

- Utilisation des tag-libs de Spring dans vos JSP
- Form :
- <%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
  - ▶ Permet de remplacer le formulaire et tous les input afin que Spring puisse les remplir/vider automatiquement

```
<form:form action="utilisateur.smvc" method="post" commandName="utilisateurBean">

    <form:input path="nom" />
    <form:input path="prenom" />
    <input type="submit" value="Valider" />

</form:form>
```

- ❑ action = nom de l'action ⇔ @RequestMapping dans la classe Java
- ❑ commandName = nom d'un bean ⇔ clef d'un ModelMap associé dans votre classe Java
- ❑ path = référence une méthode du bean ⇔ utilisateurBean.getNom() /  
utilisateurBean.getPrenom() et utilisateurBean.setNom(valeur) /  
utilisateurBean.setPrenom(valeur)

# Gestion des erreurs

43

- Pour la validation de vos formulaires vous pouvez faire usage des Validator
- De cette manière vos beans qui représentent vos formulaires restent simples
- C'est à vous de créer la classe et de la faire implémenter l'interface  
**org.springframework.validation.Validator**

# Gestion des erreurs

- org.springframework.validation.Validator aura deux méthodes
  - ➡ **public boolean supports(Class<?> pArg0)**
    - ❑ Indique les éléments validables par cette classe
  - ➡ **public void validate(Object aLoginBean, Errors errors)**
    - ❑ Fait la validation
- Pour valider l'état d'un attribut, vous pouvez faire usage des méthodes statiques qui sont dans org.springframework.validation.ValidationUtils

# Gestion des erreurs

45

- Vous ferez usage de vos Validator dans vos méthodes de contrôleurs
  - ➡ vous lappelez sur votre bean en faisant usage du paramètre BindingResult qui se charge de porter les erreurs éventuelles
- **Note :** Le validator peut aussi être un attribut de votre contrôleur
  - ➡ Vous n'êtes pas obligé de linstancier dans vos méthodes
  - ➡ Vous pouvez lannoter en @Autowired (attention à ne pas oublier de poser un @Component sur votre classe de Validator)

# Gestion des erreurs

46

```
* Soumet le formulaire. */
@RequestMapping(value = "/dologin.smvc", method = RequestMethod.POST)
public String doLogin(@ModelAttribute("loginBean") LoginBean loginBean, ModelMap modelMap,
                      BindingResult bindingResult) {
    LoginController.LOG.debug("--> Passage dans doLogin");

    LoginBeanValidator validator = new LoginBeanValidator();
    validator.validate(loginBean, bindingResult);

    if (!bindingResult.hasErrors()) {
        // Metier

        return "menu";
    }
    return "login";
}
```

# Gestion des erreurs

47

- A tout moment dans vos méthodes de contrôleur, si vous avez l'objet BindingResult sous la main, vous pouvez ajouter des messages d'erreurs :
  - Méthode reject ou addError

```
String msg = e.getMessage() != null ? e.getMessage() : e.getClass().getName();
bindingResult.reject("error.technical", new Object[] { msg }, msg);
HistoriqueController.LOG.error("Erreur:", e);
```

# Gestion des erreurs

- Dans la page JSP vous pouvez récupérer vos erreurs via la balise `errors` :

```
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="springForm"%>
<springForm:errors path="login" cssClass="error" />
```

- Si il y a un message d'erreur associé à la clef login : alors, dans la page JSP, un bloc de `<span>` sera créé avec la classe `error`
- Vous pouvez indiquer « \* » dans path pour lister toutes les erreurs
- Cette balise gère en automatique l'internationnalisation
  - Il faut juste y penser en début de projet

# Gestion des erreurs

49

- Vous pouvez aussi utiliser la balise *hasBindErrors* pour savoir si il y au moins une erreur sur le bean visé

```
<%@ taglib uri="http://www.springframework.org/tags" prefix="springTags"%>
<springTags:hasBindErrors name="virementBean">...</springTags:hasBindErrors>
```

# Gestion de l'internationnalisation

50

- Pour que l'internationnalisation fonctionne, vous devez déclarer un fichier properties sous l'id **messageSource**
- Cette déclaration se fait dans le fichier de configuration du DispatcherServlet

```
<bean id="messageSource" class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
    <property name="basename" value="classpath:/message"/>
    <property name="defaultEncoding" value="utf-8"/>
</bean>
```

# Gestion de l'internationnalisation

51

- Dans l'exemple, le fichier properties portera comme nom :
  - ▶ message\_[Locale].properties
  - ▶ Ex : message\_fr\_FR.properties pour le français de France
- Il contiendra toutes les clefs (code en Spring) pour les messages

```
9 error.wrong.password=Mauvais mot de passe
10 error.user.unknown=Utilisateur inconnu
11 error.technical=Erreur technique {0}
12 error.user.empty=Login vide
13 error.password.empty=Mot de passe vide
```

# Gestion de l'internationnalisation

52

- Dans votre JSP vous pouvez accéder à un message via la balise *message*

```
<%@ taglib uri="http://www.springframework.org/tags" prefix="springTags"%>
<springTags:message code="votre.clef" />
```

- Vous pouvez aussi y inclure des éléments de JSTL
  - <springTags:message code="\${monbean.code}" />

# Validation JSR 303

53

- En plus des Validator (ou à la place), vous pouvez faire usage de la norme Java JSR 303 pour valider vos beans
  - ➡ <http://beanvalidation.org/1.0/spec/>
- Objectifs de cette norme :
  - ➡ Proposer une spécification relative à la validation de données dans les applications Java
  - ➡ Fournir une API qui soit indépendante d'une architecture
  - ➡ Etre utilisable dans toutes les couches Java d'une application
    - Vous pouvez l'utiliser pour vos Bean, vos entités, ...
  - ➡ Standardiser la déclaration des contraintes en privilégiant **les annotations**
  - ➡ Etre facile à utiliser et extensible

# Validation JSR 303 - Annotations

54

- Les annotations associées à cette norme peuvent s'appliquer sur :
  - La classe (ou interface)
    - Utilisé quand il faut manipuler plusieurs attributs de la classe en même temps afin de réaliser la validation
  - L'attribut
  - La méthode getXxx
- Toutes les annotations seront issues du package javax.validation

# Validation JSR 303 - Annotations

55

Annotation	Description	Exemple
<b>@AssertFalse</b>	La valeur de l'attribut doit être false	<code>@AssertFalse boolean isUnsupported;</code>
<b>@AssertTrue</b>	La valeur de l'attribut doit être true	<code>@AssertTrue boolean isActive;</code>
<b>@DecimalMax</b>	La valeur doit être un chiffre à virgule inférieure ou égale à la valeur indiquée	<code>@DecimalMax("30.00") BigDecimal discount;</code>
<b>@DecimalMin</b>	La valeur doit être un chiffre à virgule supérieure ou égale à la valeur indiquée	<code>@DecimalMin("5.00") BigDecimal discount;</code>
<b>@Digits</b>	La valeur doit être un chiffre. Integer = nombre total de chiffre, fraction = nombre de chiffre après la virgule	<code>@Digits(integer=6, fraction=2) BigDecimal price;</code>
<b>@Future</b>	La valeur doit être une date dans le futur.	<code>@Future Date eventDate;</code>

# Validation JSR 303 - Annotations

56

Annotation	Description	Exemple
<b>@Max</b>	La valeur doit être un chiffre entier inférieure ou égale à la valeur indiquée	<code>@Max(10) int quantity;</code>
<b>@Min</b>	La valeur doit être un chiffre entier supérieure ou égale à la valeur indiquée	<code>@Min(5) int quantity;</code>
<b>@NotNull</b>	La valeur ne doit pas être nulle	<code>@NotNull String username;</code>
<b>@Null</b>	La valeur doit être nulle	<code>@Null String unusedString;</code>
<b>@Past</b>	La valeur doit être une date dans le passé	<code>@Past Date birthday;</code>
<b>@Pattern</b>	La valeur doit respecter l'expression régulière.	<code>@Pattern(regexp="\\"(\d{3})\\")\d{3}-\d{4}") String phoneNumber;</code>
<b>@Size</b>	Fonctionne sur des valeurs de type String, Collection, Map, tableaux et permet de valider sa taille.	<code>@Size(min=2, max=240) String briefMessage;</code>

# Validation JSR 303 - Spring

57

- Spring supporte la norme standard
- En Spring MVC elle s'utilisera avec l'annotation :
  - ▶ @Valid (ou @Validated) qui se place sur un paramètre de méthode d'un contrôleur
  - ▶ **Important** : l'attribut BindingResult **DOIT** être celui qui suit le bean à valider (sinon l'erreur sera un HTTP Status de 400)
- Les messages d'erreur, si il y en a, iront dans le binding result automatiquement.
  - ▶ Le code du message sera :
    - **NomAnnotation.nomDeLobjetVise.nomAttribut**
      - Vous pouvez retrouver cette information en debuggant le contenu du bindingresult

# Validation JSR 303 - Spring

58

- En Spring 3, pour que cela fonctionne, vous devrez aussi ajouter une dépendance vers un framework de validation
  - Généralement celui d'hibernate

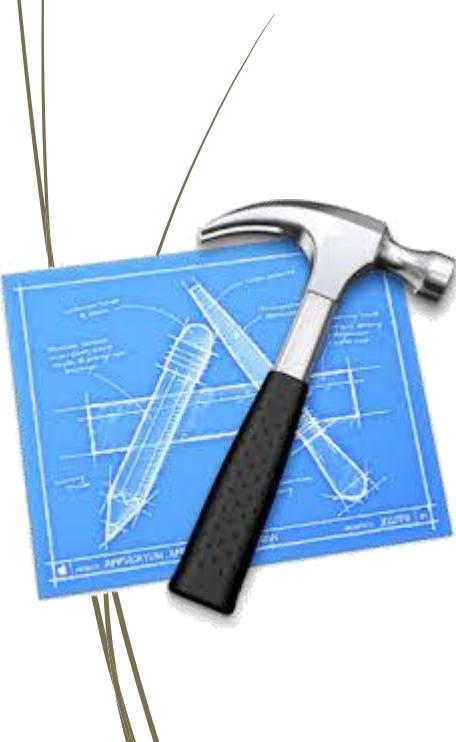
```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>4.2.0.Final</version>
</dependency>
```

Attention : la version n'est pas la même qu'hibernate

# Travaux pratiques

59

- Réaliser les travaux pratiques suivants:
  - ➡ Un projet web en Spring MVC + Hibernate



# Spring MVC

# Web Service REST

60

- ✓ Présentation
- ✓ Rappels
  - ✓ Mapping XML
  - ✓ Mapping JSON
- ✓ Norme JAX
- ✓ Framework JSON
- ✓ Rappels :
  - ✓ HTTP
  - ✓ Commandes Get, Put, ...
  - ✓ Un web service REST
  - ✓ Problématique d'authentification
  - ✓ Problématique de documentation
- ✓ Annotations Spring MVC
- ✓ Gestion des erreurs
- ✓ HATEOAS
- ✓ Junit et le Spring MVC
- ✓ Spring et Ajax

# Mapping

➤ **Définition** : prendre un objet pour le transformer en flux et inversement, prendre un flux pour le transformer en objet

❑ **Mapping XML** : Transformer un objet en flux XML et/ou un flux XML en objet

❑ **Mapping JSON** : Transformer un objet en flux JSON et/ou un flux JSON en objet

# Rappels XML

- XML : eXtensible Markup Language
  - ❑ Langage de balisage
  - ❑ Extensible
- Un langage initialement destiné à publier de l 'information et à échanger des données sur le Web
- Une recommandation du W3C (World Wide Web Consortium)
- Né dans les années 1996, adopté officiellement en octobre 2000

# Rappels XML

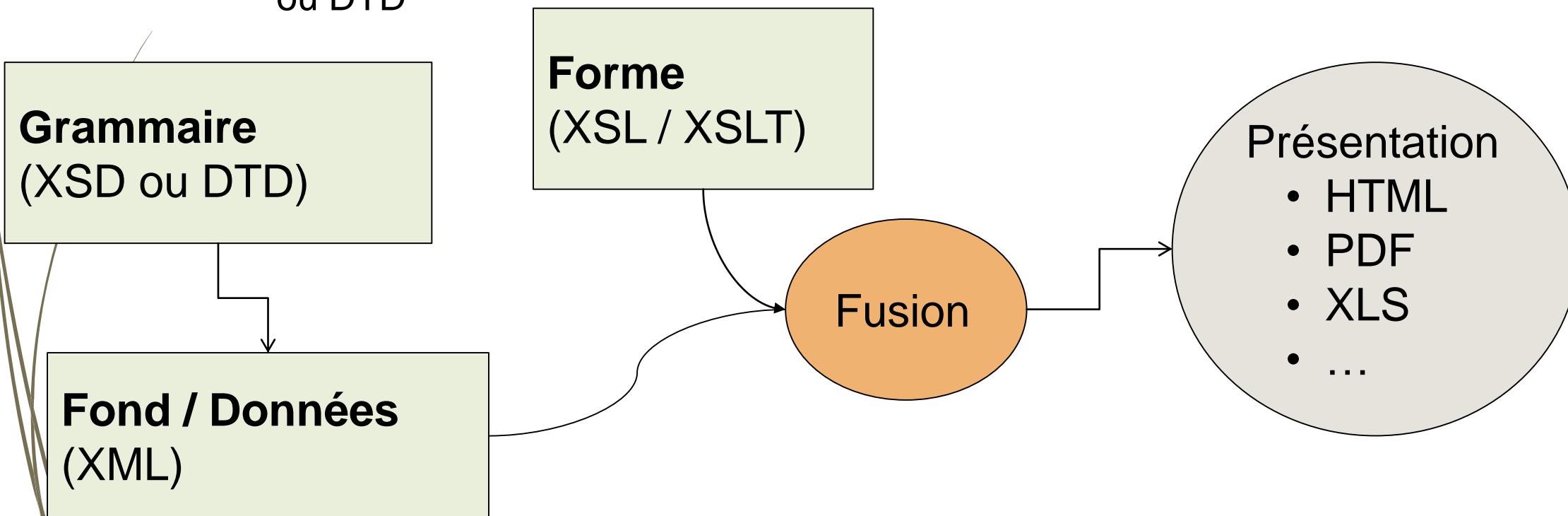
- Absence de balises prédéfinies
  - ❑ C'est à vous de choisir les balises et les attributs
- Un méta-langage permettant de prédéfinir
  - ❑ des jeux de balises
  - ❑ des règles d'utilisation pour les jeux de balises
- Rendre les données exploitable par diverses applications
  - ❑ Présentation à des utilisateurs
  - ❑ Manipulation et traitements automatisés
  - ❑ Echanges entre applications

# Exemple XML

```
<Catalogue saison="2001-2002">
  <formation filiere="internet">
    <animateur nom1="Barone"/>
    <titre>XML - les fondamentaux</titre>
    <duree>1 jour</duree>
    <type>Concepts</type>
    <prix>500e HT</prix>
  </formation>
  <formation>
    ...
  </formation>
</Catalogue>
```

➤ XML sépare le fond de la forme

- Le fond définit le contenu, les données, transmises (XML).
- La forme est fournie par un langage de style (XSL,CSS) ou du Java, .NET, JS, ....
- La grammaire définit la structure des données, elle est définie dans un fichier XSD ou DTD



# Structure du document XML

- Un document XML est structuré au moyen de balises
  - Délimitées par les symboles '<' et '>'
    - Exemple : <balise>
  - Refermées par la même balise commençant par '</>'
    - Exemple : </balise>
  - Son contenu est situé entre ces deux délimiteurs
  - Obéissant à une syntaxe stricte
- Un document XML se compose
  - D'un prologue, optionnel, mais recommandé. On y trouve divers types de déclarations (doctype, version, encoding ...)
  - D'un arbre d'éléments (les balises)
  - De commentaires, d'instructions de traitement, facultatifs et pouvant apparaître dans les deux parties précédentes

# Structure du document XML

## ➤ Les documents **bien-formés**

- ▶ Obéissent aux règles syntaxiques de XML. Ils sont exploitables par un parser.
- ▶ Sont conformes, mais pas forcément élégants ou pratiques !

## ➤ Les documents **valides**

- ▶ Sont toujours bien formés
- ▶ Respectent le modèle de document explicité dans une DTD ou XSD
- ▶ Peuvent être utilisés sous forme de simples documents bien-formés (navigateur, ...)

## ➤ Il existe des analyseurs syntaxiques validant et non validant

# Le document XML

## ➤ Références

### ► A des caractères

- Ce sont des références directes à des valeurs dans des jeux de caractères
- Elles sont considérées comme du character data
- Elles sont utiles si vous ne pouvez pas taper le caractère souhaité

### ► A des entités prédéfinies

- &lt;, &gt;, &amp;, &apos;, &quot;
- resp. <, >, &, ', "

```
<Dpt>Etudes &#38; Conseils Technologiques</Dpt>
<Dpt>Etudes &x003c; Conseils Technologiques</Dpt>
<Dpt>Etudes &amp; Conseils Technologiques</Dpt>
```

# Le document XML

## ➤ Eléments ou balise

- Ce sont des containers (`<tag></tag>`)
- Ils peuvent s'imbriquer pour former un arbre (père / enfants)
- Ils peuvent être vides (`<tag />`)

```
<Catalogue saison="2001-2002">
  <formation filiere="internet">
    <animateur nom1="Barone"/>
    <titre>XML - les fondamentaux</titre>
    <duree>1 jour</duree>
    <type>Concepts</type>
    <prix>500e HT</prix>
  </formation>
</Catalogue>
```

# Le document XML

## ➤ Attributs

- ▶ Ce sont les caractéristiques d'un élément
- ▶ Ils sont de la forme : nom = valeur
- ▶ Leur utilisation doit être "élégante" (concise et adaptée au traitement)

```
<Catalogue saison="2001-2002">
    <formation filiere="internet">
        <animateur nom1="Barone"/>
        <titre>XML - les fondamentaux</titre>
        <duree>1 jour</duree>
        <type>Concepts</type>
        <prix>500e HT</prix>
    </formation>
</Catalogue>
```

# Le document XML

## ➤ Commentaires

- Ce sont des balises silencieuses <!-- -->

```
<Catalogue saison="2001-2002">
    <!-- Mois de début et mois de fin ? -->
    <formation filiere="internet">
        <animateur nom1="Barone"/>
        <titre>XML - les fondamentaux</titre>
        <duree>1 jour</duree>
        <type>Concepts</type>
        <prix>500e HT</prix>
    </formation>
    <formation>
        ...
    </formation>
</Catalogue>
```

# XML-Schema (XSD)

- Catégorie
  - Définition de modèle de document
- Statut
  - Recommandation (2 mai 2001)
- Objectif
  - Dépasser les limitations des DTD en exprimant la structure et les contraintes sur les types de données du document sous forme XML

# XML-Schema (XSD)

- XML-Schema est un langage XML qui :
  - ▶ Se substitue aux DTD (non XML) pour définir la structure du document
  - ▶ Étend les DTD concernant les types et formats de données et gère les espaces de noms
  - ▶ Introduit la notion d'héritage entre éléments XML.
- Plusieurs tentatives précédentes
  - ▶ DCD
  - ▶ XML-Data

# XML-Schema (XSD)

## ➤ Principes

- Une description de type de document XML écrit en XML-Schema est un document XML dont
  - L'élément document est <schema>
  - Indique sa propre URI (gère les versions)
  - Définit les types d'éléments, d'attributs, de données
- Les types de données XML-Schema peuvent être
  - Des types prédéfinis
    - Types XML 1.0 (DTD) : ID, ENTITY, NOTATION, ...
    - string, binary, boolean, number, integer, decimal, real, dateTime, date, time, timePeriod, uri
  - Des types de données "utilisateur"

## XML-Schema (XSD)

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="http://tempuri.org/PurchaseOrderSchema.xsd"
    targetNamespace="http://tempuri.org/PurchaseOrderSchema.xsd"
    elementFormDefault="qualified">

    <xsd:element name="PurchaseOrder" type="tns:PurchaseOrderType"/>
    <xsd:complexType name="PurchaseOrderType">
        <xsd:sequence>
            <xsd:element name="ShipTo" type="tns:USAddress" maxOccurs="2"/>
            <xsd:element name="BillTo" type="tns:USAddress"/>
        </xsd:sequence>
        <xsd:attribute name="OrderDate" type="xsd:date"/>
    </xsd:complexType>

    <xsd:complexType name="USAddress">
        <xsd:sequence>
            <xsd:element name="name" type="xsd:string"/>
            <xsd:element name="street" type="xsd:string"/>
            <xsd:element name="city" type="xsd:string"/>
            <xsd:element name="state" type="xsd:string"/>
            <xsd:element name="zip" type="xsd:integer"/>
        </xsd:sequence>
        <xsd:attribute name="country" type="xsd:NMTOKEN" fixed="US"/>
    </xsd:complexType>
</xsd:schema>
```

# JAX

- La norme JAX : JAva Xml
- Regroupe tout ce qui concerne
  - La gestion du XML (mapping, parsing)
  - La gestion des web services Soap
  - La gestion des web services REST
- Composé d'annotations standards
  - Certaines peuvent nécessiter un framework (en plus du JDK)

- JAX-B permet de mapper des objets en XML et inversement (**Binding**)
- Le **Binding** se fait dans les classes Java avec les annotations
- Il ne nécessite pas de framework supplémentaire (depuis Java 6)
  - ➡ Attention cependant, ne sera pas opérationnel sur Android par exemple.

# JAX-B : Annotations

- **@XmlRootElement** : définit le nom de la balise root de votre flux XML. A définir sur une classe Java.
- **@XmlAttributeType** : définit comment JAX-B doit trouver les balises/attributs. A définir sur une classe Java.
- **@XmlAttribute** : définit l'information comme un attribut de votre balise XML. A définir sur une méthode get/set ou un attribut de classe Java.

# JAX-B : Annotations

79

- **@XmlElement** : définit l'information comme une sous balise de votre balise XML. A définir sur une méthode get/set ou un attribut de classe Java.
- **@XmlElementWrapper** : dans le cas des informations multi valuées (liste, map, tableau), permet de définir une balise parentes englobants les informations. A définir sur une méthode get/set ou un attribut de classe Java.
- **@XmlJavaTypeAdapter(UneClasse.class)** : Ou UneClasse hérite de javax.xml.bind.annotation.adapters.XmlAdapter, permet de définir comment transformer les informations. A définir sur une méthode get/set ou un attribut de classe Java.

# JAX-B : Annotations

80

## ➤ Exemples

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<adresse ville="Versailles"
           codePostal="78000"
           pays="France">
    ...
    <rue>Pave des gardes</rue>
</adresse>
```

```
package fr.xml;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;

/**
 * Classe qui represente une adresse. <br/>
 */
@XmlRootElement(name = "adresse")
@XmlAccessorType(XmlAccessType.NONE)
public class Adresse {
    @XmlAttribute
    private String ville;
    @XmlAttribute
    private int codePostal;
    @XmlElement
    private String rue;
    @XmlAttribute
    private String pays;

    /**
     * Constructeur.
     */
    public Adresse() {
        super();
    }
}
```

# JAX-B : Fabriquer un flux

- L'écriture se réalise grâce à un **Marshaller**
- L'objet **JAXBContext** se charge d'analyser les classes annotées
  - Attention à ne pas oublier de classe
- De lui on peut obtenir un **Marshaller**

```
// Creation du fichier XML
try {
    JAXBContext jaxbContext = JAXBContext.newInstance(Personne.class, Adresse.class);
    Marshaller jaxbMarshaller = jaxbContext.createMarshaller();
    jaxbMarshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
    jaxbMarshaller.marshal(adr01, new File(fichierSortie));
} catch (Exception e) {
    e.printStackTrace();
}
```

# JAX-B : Lire un flux

82

- La lecture se réalise grâce à un **Unmarshaller**
- L'objet **JAXBContext** se charge d'analyser les classes annotées
  - ▶ Attention à ne pas oublier de classe
- De lui on peut obtenir un **Unmarshaller**

```
// Lecture du fichier
try {
    JAXBContext jaxbContext = JAXBContext.newInstance(Personne.class, Adresse.class);
    Unmarshaller jaxbUnmarshaller = jaxbContext.createUnmarshaller();
    Personne personneLue = (Personne) jaxbUnmarshaller.unmarshal(new File(fichierEntree));
    System.out.println(personneLue);
} catch (Exception e) {
    e.printStackTrace();
}
```

# JAX-B : JAXBContext

83

- L'objet **JAXBContext** se charge d'analyser les classes annotées
- Vous pouvez lui indiquer, soit :
  - ➡ Les classes annotées séparées par des virgules (des objets `java.lang.Class`)
  - ➡ Les packages contenant des classes annotées, aussi séparés par des virgules (un objet `java.lang.String`)

# JAX-B : XmlAdapter

- Quand votre attribut est de type complexe (une **date** par exemple) vous devrez créer un objet qui va expliquer comment faire la transformation XML <-> Valeur
- Cette objet sera une classe Java qui implémentera l'interface  
**javax.xml.bind.annotation.adapters.XmlAdapter**
- Cette classe **générique** prend deux informations
  - ❑ La première le type de sortie (en général une String)
  - ❑ Le seconde sera le type de votre attribut

# JAX-B : XmlAdapter

85

- Exemple : ici on a un adapter pour les dates dans le format de notre choix
- <Sortie, Entrée>

```
package fr.xml;

import java.text.SimpleDateFormat;
import java.util.Date;

import javax.xml.bind.annotation.adapters.XmlAdapter;

public class DateAdapter extends XmlAdapter<String, Date> {

    @Override
    public Date unmarshal(String aV) throws Exception {
        if (aV == null || aV.trim().isEmpty()) {
            return null;
        }
        SimpleDateFormat dateFormat = new SimpleDateFormat("dd/MM/yyyy");
        dateFormat.setLenient(false);
        return dateFormat.parse(aV);
    }

    @Override
    public String marshal(Date aV) throws Exception {
        if (aV == null) {
            return null;
        }
        SimpleDateFormat dateFormat = new SimpleDateFormat("dd/MM/yyyy");
        dateFormat.setLenient(false);
        return dateFormat.format(aV);
    }
}
```

# JAX-B : XmlAdapter

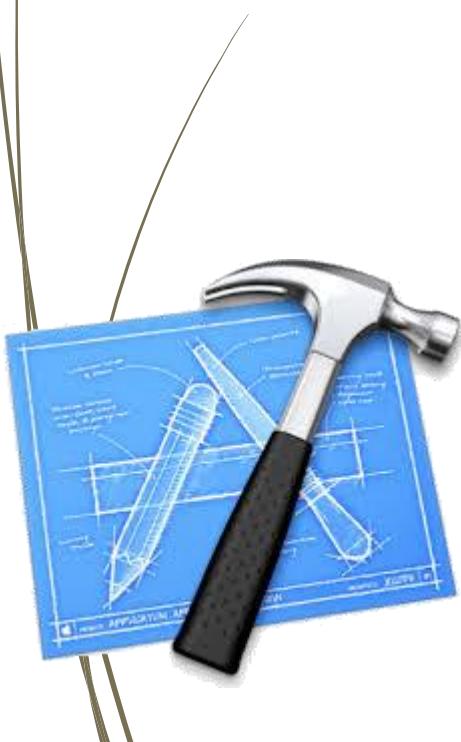
- Pour utiliser votre adapter il suffit d'utiliser l'annotation **@XmlJavaTypeAdapter** sur l'attribut ou la méthode get/set

```
@XmlElement  
@XmlJavaTypeAdapter(DateAdapter.class)  
private Date dateNaissance;
```

# Travaux pratiques

87

- Réalisons un Mapping XML de nos entités



- L'API Java pour le traitement XML permet aux applications d'analyser et transformer des documents XML en utilisant une API qui est indépendant d'une implémentation d'un processeur XML particulier. (**Parser**)
- JAX-P permet aussi de brancher/débrancher une implémentation de processeur XML simplement
- Toute application qui fait de la gestion de flux XML devrait se limiter à l'API JAXP et éviter l'utilisation d'API spécifique.
- JAXP sait très bien faire du
  - ❑ DOM : traitement global du flux, notion de chemin et de liens entre balises
  - ❑ SAX : traitement au fur et à mesure

# JAX-P

- JAXP sait moins bien faire les transformations XSLT
  - ➡ A vous de voir en fonction de vos besoins de transformation
- Avant il fallait ajouter à ses projets XML des librairies comme Xerces ou Xalan
- Depuis le Java 5, une implémentation représenté par JAX-P est disponible dans le JDK
  - ➡ StAX

# JAX-WS

- JAX-WS représente les annotations permettant de réaliser des Web services SOAP
  - Package javax.jws.\*
- [https://docs.oracle.com/cd/E13222\\_01/wls/docs103/webserver\\_ref/annotations.html](https://docs.oracle.com/cd/E13222_01/wls/docs103/webserver_ref/annotations.html)
- Pour réaliser un projet en SOAP, il faut choisir un framework qui respecte cette norme
  - Apache AXIS / AXIS2
  - Apache CXF
  - gSOAP

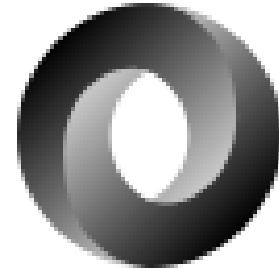
# JAX-RS

91

- JAX-RS représente les annotations permettant de réaliser des Web services REST
  - ▶ Package javax.ws.rs
- <http://docs.oracle.com/javaee/6/tutorial/doc/gilik.html>
- Pour réaliser un projet en REST, il faut choisir un framework qui respecte cette norme
  - ▶ Jersey
  - ▶ Spring MVC
  - ▶ Restlet

# Rappel JSON

- JSON : JavaScript Object Notation
  - Format de données
- Né dans les années 2002-2005, normalisé par IETF (Internet Engineering Task Force)
- Le JSON sert exclusivement à représenter des données



# JSON vs XML

## ➤ Contrairement au XML en JSON

- ❑ Il n'y a pas de balise
- ❑ Il n'y a pas d'attribut
- ❑ Il n'y a pas de grammaire (DTD, XSD)

## ➤ Avantages

- ❑ Flux plus petit qu'en XML
- ❑ Structuration non contraignante et donc simple
- ❑ Lu très simplement par les librairies JavaScript

## ➤ Inconvénients

- ❑ Structuration non contraignante et donc non valide
- ❑ Type des données limité

# Flux JSON

94

- La structure du JSON se limite à deux représentation
  - ▶ Un objet JSON : { ... }
  - ▶ Un tableau d'objets JSON : [ {...} , {...}, {...}]
- Un objet JSON est composé par
  - ▶ Des noms de propriétés entre " " (ou ' )
  - ▶ Des valeurs dont le type peut être :
    - ❑ Chaîne de caractères (entre " " ou ' )
    - ❑ Nombre (entier ou flottant)
    - ❑ Un objet Json
    - ❑ Boolean (true ou false)
    - ❑ Un tableau ([ ])
    - ❑ La valeur null

# Exemple : un objet JSON

```
{  
    "lastName": "Gillet",  
    "lastLogin": 1421766774000,  
    "address": {"office": "Etage 1, Porte 281"},  
    "firstLogin": 1421766066000,  
    "sex": 0,  
    "creationDate": 1409750532000,  
    "login": "testuser98"  
}
```

# JSON : contraintes syntaxiques

96

- Le ':' sépare le nom de la propriété de sa valeur
- L'usage des " sur le nom des propriétés n'est pas une option
  - ➡ "lastName" et surtout pas ~~lastName~~ (éviter 'lastName')
- La ',' permet de séparer chaque propriété entre elle
  - ➡ Elle n'est pas obligatoire sur le dernier élément

# Exemple un Objet : JSON

- Ceci est un Objet JSON

```
{  
    "employees": [  
        {"firstName":"John", "lastName":"Doe", "sex":true },  
        {"firstName":"Anna", "lastName":"Smith", "sex":true },  
        {"firstName":"Peter", "lastName":"Jones", "sex":false }  
    ],  
    "taille" : 3  
}
```

- La valeur de la propriété *employees* est représentée par un tableau d'objet JSON.

# JSON et Java

98

- Il n'y a pas pour le moment d'annotation **officielle** comme en JAX-B
- Cependant, vous avez un lien entre JAX-B et JSON
- Vous pouvez faire usage des Marchaller et Unmarchaller de JAX-B afin de produire/lire du JSON
- <https://docs.oracle.com/middleware/1212/toplink/TLADG/json.htm#TLADG1144>

# JSON et Java

99

- L'approche Oracle actuelle est malheureusement assez lourde
- En général, on fait plutôt usage d'un framework extérieur :
  - ❑ <https://jsonp.java.net>
  - ❑ <http://json-lib.sourceforge.net/>
  - ❑ <http://flexjson.sourceforge.net/>
  - ❑ <https://github.com/google/gson>
  - ❑ <https://github.com/FasterXML/jackson-core>

# JSON et Java

- Le plus utilisé est **Jackson**
  - ▶ Le plus rapide
  - ▶ Possède des annotations (propriétaires)
  - ▶ Peut aussi faire usage des annotations JAX-B (pratique si vous avez besoin de mapping XML et JSON)
  - ▶ Fonctionne de manière transparente et automatique avec de nombreux framework (Spring, Jersey, ...)
  - ▶ **MAIS** : ne marche pas sous Android avec pro-guard

# Mapping

- En général, le mapping respectera les contraintes suivantes
  - Web service SOAP (en JAX-WS)
    - XML (via JAX-B)
  - Web Service REST (en JAX-RS) :
    - XML (via JAX-B)
    - JSON (via un framework)

# Travaux pratiques

102

- Réalisons un Mapping JSON de nos entités



# Web Service : Présentation

103

## ➤ Caractéristiques du service web :

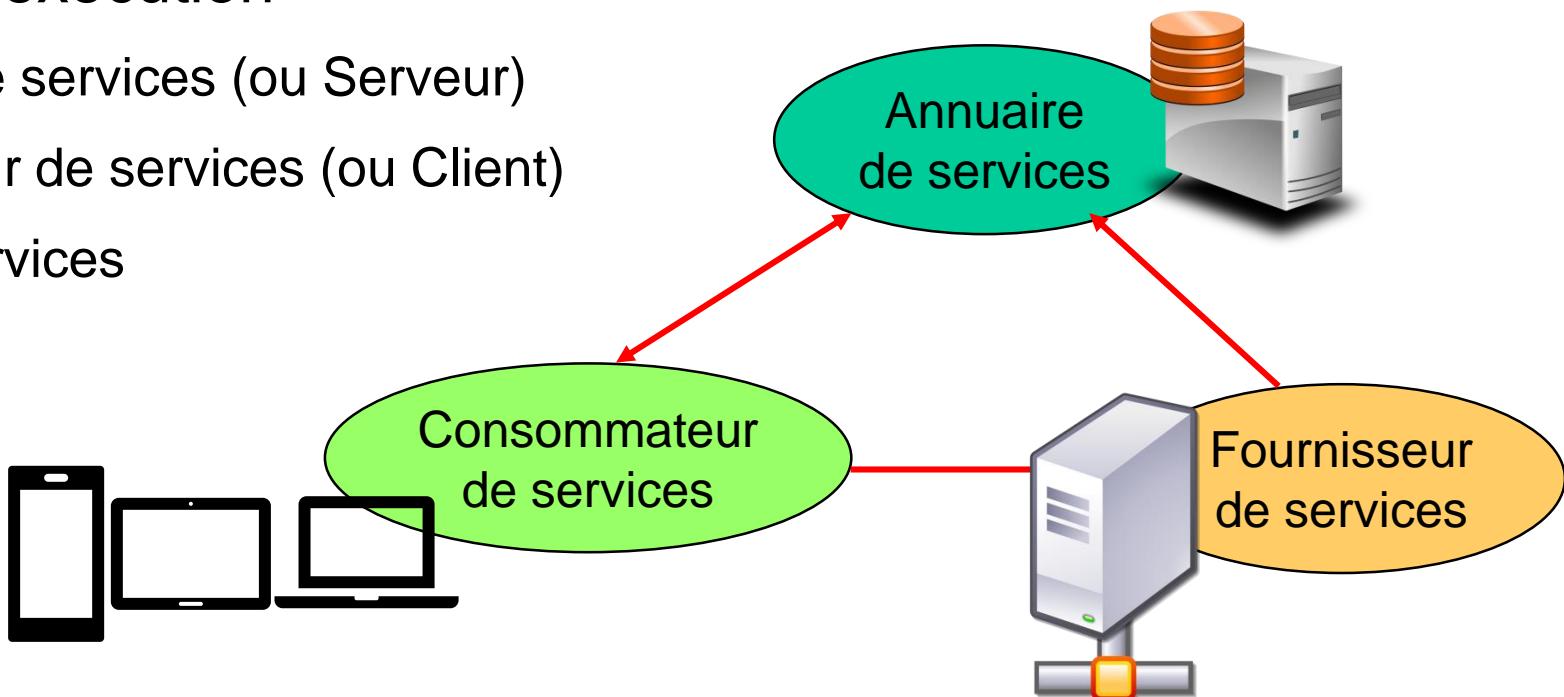
- ▶ **Autonome** : L'implémentation d'un service doit être la plus indépendante possible vis-à-vis du parc de services existant.
- ▶ **Localisable** : Un service doit être formellement localisable. Dans les standards des services web (WS-\*), la localisation se définit à l'aide d'une URI.
- ▶ **Isolé** : La logique de communication doit être isolée de la logique métier. Une évolution du protocole de communication ne doit impacter ni la signature du service ni son implémentation.
- ▶ **Sans état** : L'invocation courante du service doit être indépendante de son invocation précédente.
- ▶ **Transparent** : L'exécution d'un service doit être facilement diagnostiquée (état des variables et des threads, performances ...).

# Web Service : Présentation

104

## ➤ Architecture et rôle

- ▶ Les Web services sont construits sur le concept de l'informatique distribuée
- ▶ Une architecture Web Services consiste en trois rôles ou environnement d'exécution
  - ❑ Le fournisseur de services (ou Serveur)
  - ❑ Le consommateur de services (ou Client)
  - ❑ L'annuaire de services



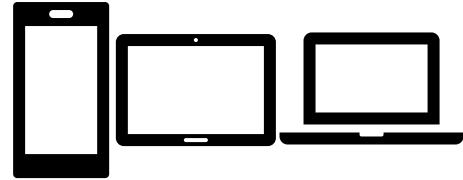
# Web Service : Présentation



## ➤ Le fournisseur de services (ou Serveur)

- Il exécute le Web Service et fournit l'accès à celui-ci au travers d'une interface standard
- Il est séparé en deux entités
  - ❑ L'interface du service : elle fournit le contrat du service (ce qu'on peut utiliser) aux applications utilisatrices
  - ❑ L'implémentation du service : elle fournit l'implémentation de cette interface
    - Exemple : un composant hébergé dans un serveur d'application

# Web Service : Présentation



- Le consommateur de services (ou Client)
  - Il s'agit de l'application utilisatrice du Web Service
  - Il établit une liaison avec l'interface du service proposée par le fournisseur de services
  - Puis il utilise le service souhaité
- Un consommateur est donc un programme qui sait envoyer des requêtes HTTP et recevoir des flux

# Web Service : Présentation

107

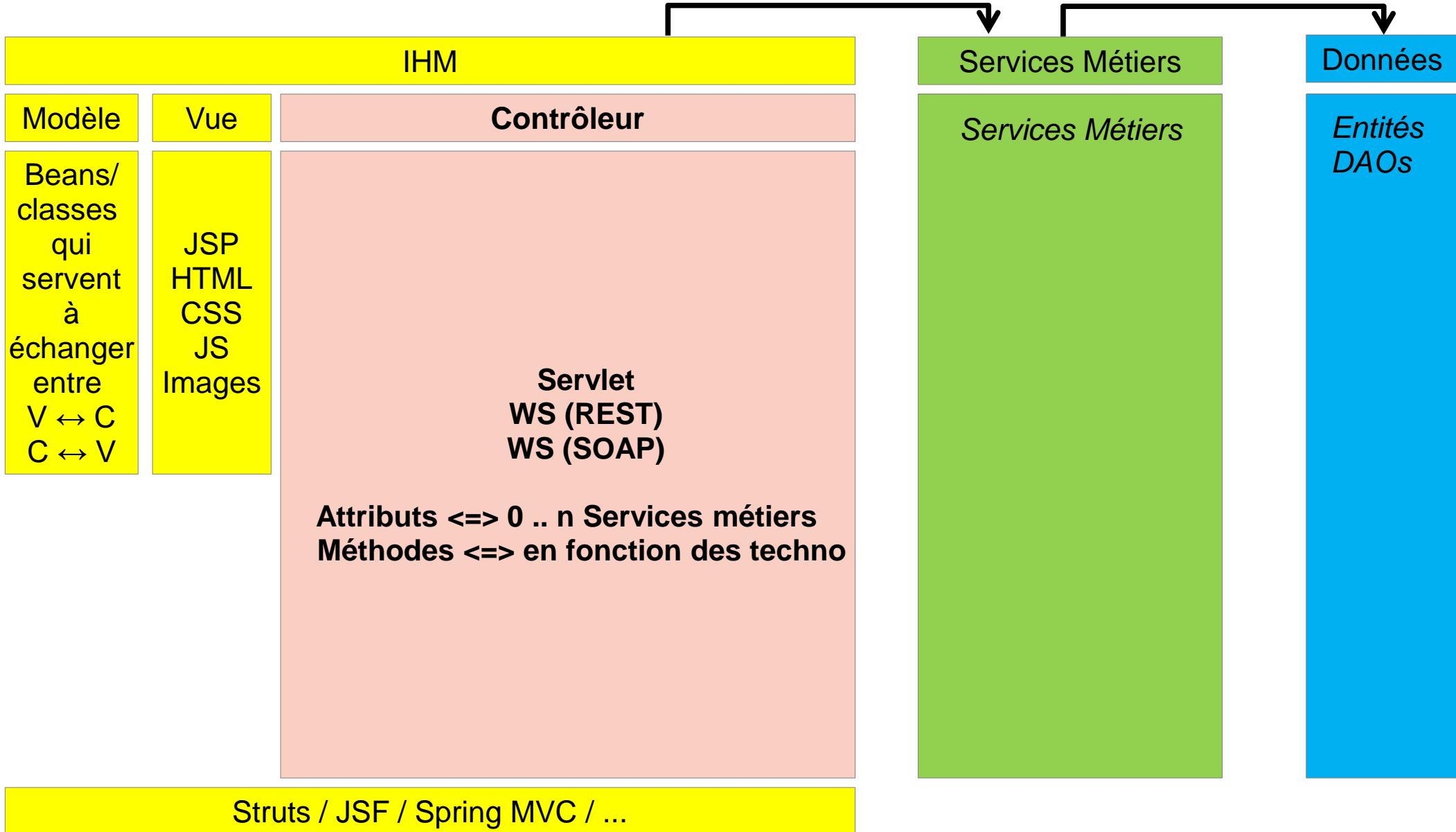


## ➤ L'annuaire de services

- ▶ Il fournit le moyen de trouver les Web Services publiés par les fournisseurs de services
- ▶ Principes
  - ❑ Le fournisseur de services publie ses services dans l'annuaire
  - ❑ Le consommateur de services s'adresse à l'annuaire pour obtenir des informations sur les services disponibles
  - ❑ A PROBTP il s'agit de l'application MARS

# Web Service : Présentation

108

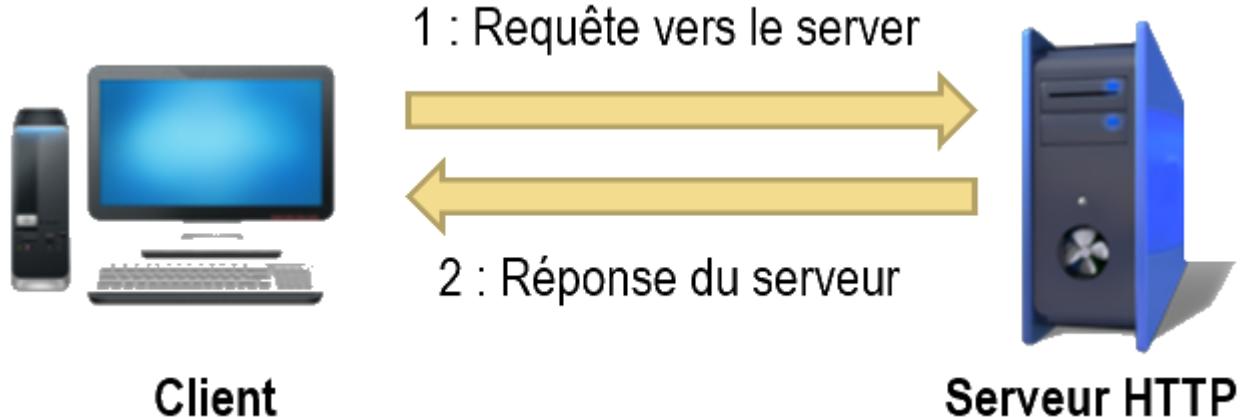


# Le protocole HTTP

➤ Un protocole est une convention à respecter pour bien communiquer.

► HTTP :

- ❑ Hypertext Transfer Protocol soit protocole de transfert hypertexte
- ❑ Développé pour le World Wide Web
- ❑ Il fait parti de la couche "application"
- ❑ Son port par défaut est le port 80 (443 pour le https)



# HTTP : Communication

110

- Le HTTP permet d'échanger des flux textes entre un client et un serveur
- Le client fait une requête (= une demande) au serveur.  
Ces demandes peuvent être de plusieurs types utilisant ce que l'on appelle "méthodes" / "Commandes"
  - ▶ get : Demander une ressource au serveur (exemple: un fichier, une page html ou des informations).
  - ▶ post : Envoyer des données vers le serveur (exemple : enregistrer un formulaire, envoyer un message)
  - ▶ put : Créer ou modifier une ressource
  - ▶ delete : Supprimer des ressources
  - ▶ head : Demande d'information sur une ressource.
- Le serveur exécute la demande et répond
  - ▶ Un flux
  - ▶ Un HTTP Status

# Le protocole HTTP : requête

- La requête est divisée en deux blocs :
- L'entête qui contiendra des informations sur la méthode employée, le serveur et le client lui-même

## En-tête GET

```
GET /chemin/hello.htm HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
Host: www.site.com
```

## En-tête POST

```
POST /chemin/connexion HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
Host: www.site.com
```

- Le corps contient les données envoyées

## Corps POST

```
Login: Jhon
Password: 1234
```

# HTTP : GET

112

- C'est la méthode la plus courante pour demander une ressource.
- Une requête **GET** est sans effet sur la ressource, elle ne modifie pas cette dernière
- On peut utiliser le **GET** quand on souhaite réaliser un select, insert, update, delete
- On fait un **GET** quand
  - ▶ On clique sur un lien hypertext
  - ▶ On soumet un formulaire qui est en méthode GET
  - ▶ Dans ce cas, les paramètres **seront visibles** sur l'URL

# HTTP : POST

113

- Cette méthode est utilisée pour transmettre des données en vue d'un traitement à une ressource.
- Une requête **POST** entraîne généralement un traitement côté serveur.
- On peut utiliser le **POST** quand on souhaite réaliser un select, insert, update, delete
- On fait un **POST** quand
  - ➡ On soumet un formulaire qui est en méthode POST
  - ➡ Dans ce cas, les paramètres **ne seront pas visibles** sur l'URL

# HTTP : PUT

- Cette méthode permet de remplacer ou d'ajouter une ressource sur le serveur.
- On peut utiliser le **PUT** quand on souhaite réaliser un insert, update, delete
- Le **PUT** à deux avantages par rapport aux POST/GET
  - ➡ Son résultat ne sera pas mis en cache
  - ➡ Si deux requêtes PUT identiques arrivent sur le serveur, une seule sera traitée (idempotent)
- On ne peut faire du **PUT** qu'à travers du code

# HTTP : PATCH

115

- Cette méthode permet, contrairement à PUT, de faire une modification partielle d'une ressource.
- On peut utiliser le **PATCH** quand on souhaite réaliser un update
- On ne peut faire du **PATCH** qu'à travers du code
- Attention :
  - ➡ **PATCH** n'est pas idempotent
  - ➡ N'est pas supportée par tous les serveurs / framework

# HTTP : DELETE

116

- Cette méthode permet de supprimer une ressource du serveur.
- On peut utiliser le **DELETE** quand on souhaite réaliser un delete
- On ne peut faire du **DELETE** qu'à travers du code

# HTTP : OPTIONS

117

- Cette méthode permet d'obtenir les options de communication d'une ressource ou du serveur en général.
- On peut utiliser le **OPTIONS** quand on souhaite récupérer des informations
- On ne peut faire du **OPTIONS** qu'à travers du code

# HTTP : TRACE

118

- Cette méthode demande au serveur de retourner ce qu'il a reçu, dans le but de tester et effectuer un diagnostic sur la connexion.
- On peut utiliser le **TRACE** quand on souhaite récupérer des informations
- On ne peut faire du **TRACE** qu'à travers du code

# HTTP : CONNECT

119

- Cette méthode permet d'utiliser un proxy comme un tunnel de communication.
- On ne peut faire du **CONNECT** qu'à travers du code

# HTTP : HEAD

120

- Cette méthode ne demande que des informations sur la ressource, sans demander la ressource elle-même.
- On ne peut faire du HEAD qu'à travers du code

# Méthodes HTTP Récapitulatif

Method	Idempotent	Safe
OPTIONS	Oui	Oui
GET	Oui	Oui
HEAD	Oui	Oui
PUT	Oui	Non
DELETE	Oui	Non
PATCH	Non	Non
POST	Non	Non

# Méthodes HTTP

- Une méthode **idempotente** signifie que le résultat d'une requête effectuée avec succès est indépendant du nombre de fois où elle a été exécutée.
  - ▶ OPTIONS,GET,HEAD,PUT,DELETE sont *idempotent*
- Une méthode **safe** signifie que la requête ne modifie en rien l'état du serveur.
  - ▶ OPTIONS,GET,HEAD sont *safe*

# REST

- REpresentational State Transfert
- Moyen de demander des traitements ou informations à un serveur
- Repose sur le protocole HTTP (et ses méthodes GET, PUT, ...)
- En théorie sans état (en pratique cela se complique)
  - ➡ Chaque requête est indépendante

# REST sans état

- Le client demande quelque chose au serveur via une commande un URI et des paramètres
  - ▶ Paramètres WEB (GET classique)
  - ▶ Paramètres dans le HEAD ou le BODY
- Le serveur analyse sa demande et répond sous la forme d'un flux (XML / JSON / Binaire) et/ou d'un statut
- Le client et le serveur ne se connaissent plus

# REST et HTTP

125

- Lors de la réalisation de mécanique CRUD :
  - ▶ Create = **PUT**
  - ▶ Retrieve = **GET**
  - ▶ Update = **PUT** (ou **PATCH**)
  - ▶ Delete = **DELETE**
- On évite le **POST** (et souvent le **PATCH**) sur les actions d'écriture de données car il n'est pas *idempotent*
- On parle ici de recommandations, vous pouvez tout faire en **GET** si cela vous chante, c'est à **vous** de décider de la méthode HTTP à utiliser

# REST et URL

126

- Vos Web services seront disponibles sur des URL
- Pour passer de l'information vous avez trois techniques :
  1. Passer des paramètres : URL?clef=valeur&clef=valeur ...
  2. Passer des informations dynamiques sur l'URL : URL/valDynA/valDynB
  3. Passer des objets JSON lors de l'appel : URL
- En fonction de vos besoins et des frameworks, vous pouvez mixer les trois techniques
  - ❑ Ajax simple : technique 1, 2, 3
  - ❑ Angular : technique 2, 3
  - ❑ Web simple : technique (via formulaire) 1, 2

# REST et URL

127

- Exemple, imaginons que nous réalisons des web services REST sur la gestion de livres.
  - ➡ On ne parle ici que des techniques 1, 2 car la troisième n'est pas visible
- Rechercher un ou plusieurs livres (en **GET**)
  - ➡ *http://serveur/context/rest/livres?crt0=val0&crt1=val1*
  - ➡ Ex: *http://serveur/context/rest/livres?titre=terre&annee=1986*
- Supprimer un livre (en **DELETE**)
  - ➡ *http://serveur/context/rest/livres/valeurIdDuLivre*
  - ➡ Ex : *http://serveur/context/rest/livres/109*

# REST et URL

128

## ➤ Mettre à jour un livre (en **PATCH** ou **PUT**)

- ▶ *http://serveur/context/rest/livres/valIdDuLivre?col0=val0&col3=val3*
- ▶ Ex : *http://serveur/context/rest/livres/109?annee=2008&nbpage=298*

## ➤ Insérer un livre (en **PUT**)

- ▶ *http://serveur/context/rest/livres?col0=val0&col1=val1*
- ▶ Ex :  
*http://serveur/context/rest/livres?titre=S4&author=Max&annee=2016&nbpage=405*

# REST et URL

129

➤ Même exemple, mais via le 2,3  
(utilisation du JSON)

► On définit le flux JSON suivant  
d'exemple qui représentera un  
livre :

► On définit le flux JSON suivant  
d'exemple qui représentera des  
livres :

```
{  
    "id" : 103,  
    "titre" : "xxxx",  
    "annee" : 2014,  
    "nbpage" : 50,  
    "author" : "Jhon Smith"  
}
```

```
{  
    taille : 2,  
    livres : [  
        {  
            "id" : 103,  
            "titre" : "xxxx",  
            "annee" : 2014,  
            "nbpage" : 50,  
            "author" : "Jhon Smith"  
        },  
        {  
            "id" : 104,  
            "titre" : "xxxx",  
            "annee" : 2015,  
            "nbpage" : 150,  
            "author" : "Jhon Smith"  
        }]  
}
```

# REST et URL

130

## ➤ Rechercher un ou plusieurs livres (en **GET**)

- ▶ *http://serveur/context/rest/livres*
- ▶ Ex: *http://serveur/context/rest/livres*

- On passe dans le body un objet JSON de recherche qui ressemble à l'objet JSON livre mais où toutes les propriétés sont optionnelles
- On récupère un objet JSON de type **livres**

## ➤ Supprimer un livre (en **DELETE**)

- ▶ *http://serveur/context/rest/livres/valeurIdDuLivre*
- ▶ Ex : *http://serveur/context/rest/livres/109*

# REST et URL

131

- Mettre à jour un livre (en **PATCH** ou **PUT**)
  - ▶ *http://serveur/context/rest/livres/valldDuLivre*
  - ▶ Ex : *http://serveur/context/rest/livres/109*
    - On passe dans le body un objet JSON de type livre mais où toutes les propriétés sont optionnelles
    - On récupère un objet JSON de type livre
- Insérer un livre (en **PUT**)
  - ▶ *http://serveur/context/rest/livres*
  - ▶ Ex : *http://serveur/context/rest/livres*
    - On passe dans le body un objet JSON de type livre
    - On récupère un objet JSON de type livre (avec son ID)

# HTTP : Réponse

132

- La réponse tout comme la demande est composée elle aussi des deux parties, l'en-tête et le corps.

**En-tête réponse get**

```
HTTP/1.1 200 OK
Date: Fri, 06 May 2016 16:04:09 GMT
Server: Apache/2.4.6 (CentOS) PHP/5.4.16
```

**Corps réponse get**

```
<!DOCTYPE html>
<head>
<meta charset="utf-8">
</head>
<body>
...

```

# REST et les codes retours - status

133

- Quand le serveur répond, il répond un status et potentiellement un flux
- Codes commençant par :
  - ➡ 1xx : Information pour le client.
  - ➡ 2xx : Indique que la demande a été traitée avec succès
  - ➡ 3xx : Indique qu'il faut rediriger le client ou que la ressource a été déplacée.
  - ➡ 4xx : Généralement une erreur du client
  - ➡ 5xx : Le serveur a rencontré un problème de fonctionnement
- [https://fr.wikipedia.org/wiki/Liste\\_des\\_codes\\_HTTP](https://fr.wikipedia.org/wiki/Liste_des_codes_HTTP)

# REST et les codes retours - status

134

- **ATTENTION** : selon qui à fait le demande (code Java, Javascript, client web, ...) si le code retour n'est pas 200 alors il faudra tester le comportement
  - Un client peut traiter le body d'une réponse 500 alors qu'un autre ne le fera pas.
  - Ne soyez pas trop *créatif* sur l'usage des *status*

# JAX-RS

135

- JAX-RS est la norme standard Java pour réaliser des web services REST
  - ▶ Package javax.ws.rs
- <http://docs.oracle.com/javaee/6/tutorial/doc/gilik.html>
- Pour réaliser un projet en REST, il faut choisir un framework qui respecte cette norme
  - ▶ Jersey
  - ▶ Spring MVC
  - ▶ Restlet

# Problématique de l'authentification

- Si un web service REST ne conserve pas d'information, comment gérer l'authentification ?
- Scénario classique (en MVC)
  - ❑ Vue login : soumission du formulaire web qui nous fait partir vers le contrôleur d'authentification
  - ❑ Contrôleur d'authentification : Il valide les paramètres, fait l'authentification, place en session une information puis passe à l'écran suivant

# Problématique de l'authentification : S1

137

- Le web service renvoie une information (un id, un token d'authentification, ...) lors de son appel.
- Ce token est passé dans toutes les requêtes suivantes
  - ➡ `http://serveur/context/rest/livres?token=Xxx&col0=val0&col1=val1`
- Avantage
  - ➡ Très simple à développer
  - ➡ On reste en REST (sans état)
- Inconvénient
  - ➡ Lourdeur dans la gestion des URI

# Problématique de l'authentification : S2

138

- On fait usage de l'objet HttpSession dans le web service côté serveur.
- On fait comme si le web service restait une servlet
- Avantage
  - ➡ Très simple à développer
- Inconvénient
  - ➡ On ne respecte pas la norme REST (sans état)

# Problématique de l'authentification : S3

- Elle ressemble à la solution 1, mais au lieu de placer le token dans l'URL on le place dans le Header HTTP
- Avantage
  - ➡ On respecte la norme REST (sans état)
- Inconvénient
  - ➡ Pas toujours simple à développer

# Problématique de l'authentification : SX

- En fonction des implémentations de framework vous pourrez faire usage de solutions automatiques.
- Par exemple
  - ➡ <https://jwt.io/> (JWT webtoken)
- Restez simple et testez bien votre choix sur plusieurs serveurs et plusieurs clients (navigateurs, clients JS)

# Problématique de documentation

- Faire des web service c'est bien, mais il n'y pas de moyen autre que la documentation pour savoir :
  - Ce que l'on doit lui envoyer
    - paramètres, JSON, format du JSON
    - méthode HTTP
  - Ce que l'on va recevoir
    - Flux JSON
    - Status HTTP

# Problématique de documentation

- Vous pouvez faire usage de framework pour générer automatiquement et dynamiquement de la documentation sur vos web services
- Par exemple via le framework swagger
  - ➡ <https://swagger.io/>
- Attention : sa mise en place peut demander du temps

# Spring MVC - Configuration

143

- Pour configurer notre Spring afin de faire du web service :
  - ❑ Le Spring MVC est en MVC 2
  - ❑ Il faudra déclarer une servlet qui servira de contrôleur général
    - Elle se déclare dans le fichier WEB-INF/web.xml de votre application web
    - Elle est paramétrable
      - URL Mapping
      - Fichier de configuration Spring dédié
- Rappel : Spring MVC fonctionne aussi en mode JSP / servlet, pour le Spring c'est la même servlet

# Spring MVC - Configuration

144

## ➤ Dans le fichier WEB-INF/web.xml

```
<!-- Chargement des fichiers de configuration Spring -->
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/configuration/spring/bootstrap-config.xml /WEB-INF/configuration/spring/security-config.xml</param-value>
</context-param>

<!-- Ce Listener permet de recharger vos fichiers de configurations Spring -->
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>

<!-- Controleur général pour les WS -->
<!-- Si son nom = dispatcher alors il y a un fichier de configuration WEB-INF/dispatcher.xml -->
<servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>/rest/*</url-pattern>
</servlet-mapping>
```

# Spring MVC - Configuration

145

- Dans l'exemple :
  - Le  **servlet-name = dispatcher**
  - Il faudra créer un fichier de configuration Spring qui devra porter le nom  **dispatcher-servlet.xml** et se trouver dans le dossier **WEB-INF**
- Le fichier de configuration pourra
  - Etre vide si vous avez tout mis dans d'autre fichier Spring
  - Indiquer des éléments spécifiques au Spring MVC
    - Par exemple : le component-scan sur le package des contrôleurs.
- Si vous souhaitez faire du **PUT**, n'oublier pas d'ajouter le filter `org.springframework.web.filter.HttpPutFormContentFilter` dans votre `web.xml`

# Spring MVC - Configuration

146

- Exemple pour un fichier de configuration Spring MVC pour web services

```
1①<beans xmlns="http://www.springframework.org/schema/beans"
2      xmlns:context="http://www.springframework.org/schema/context"
3      xmlns:mvc="http://www.springframework.org/schema/mvc"
4      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5      xsi:schemaLocation="
6          http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
7          http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd
8          http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc.xsd">
9
10     <context:component-scan base-package="com.banque.web.controller.rest" />
11
12     <mvc:annotation-driven />
13
14 </beans>
```

# Spring MVC : **@RestController**

- Le Spring a ses propres annotations, il est conseillé de les utiliser.
  - ▶ Donc de ne pas faire usage des annotations JAX-RS
- **@RestController** est une annotation de classe
  - ▶ Elle permet d'indiquer que la classe est un web service
  - ▶ Indispensable pour le contexte Spring.
- Ne pas oublier de dire au Spring qu'il doit scanner votre package qui contient vos classes de web services
  - `context:component-scan`
- Si vous êtes en Spring < 4, vous devrez faire usage de l'annotation **@Controller**

# Spring MVC : @RestController

- ▶ On lui donne les attributs dont elle a besoin
  - 0 ou N Services métiers
  - **JAMAIS** de DAO ou de contrôleurs
- ▶ Nous sommes en Spring, les attributs seront sûrement annotés avec `@Autowired` afin que le Spring les injecte.
- ▶ Note : dans le cas des web services rest, il n'est pas nécessaire de faire une interface

# Spring MVC : @RequestMapping

- Annotation de classe et/ou de méthode
- Permet d'indiquer l'URL de votre web service
- Sur une **classe** : Indique la racine de l'URL
  - `@RequestMapping("/comptes")`
- Sur une **méthode** : s'additionne au chemin de la classe
  - `@RequestMapping(value = "/lister" ...)`
  - `/comptes/lister`

# Spring MVC : @RequestMapping

- Exemple de déclaration d'une classe Rest

```
33  @Controller
34  @RequestMapping("/authentifier")
35  public class AuthentifierRestController {
36      private static final Logger LOG = LogManager.getLogger();
37      @Autowired
38      private IAuthentificationService authenticationService;
```

- Ce contrôleur sera joignable sur l'URL
  - http(s)://[server]:[port]/[contextjee]/[urlmappingspringmvc]/[requestmappingclass]
  - http://localhost:8080/netbank/rest/authentifier
- Ce contrôleur sera lié par injection de dépendance avec notre **service métier** d'authentification

# Spring MVC : @RequestMapping

151

- @RequestMapping
  - ▶ s'additionne au chemin de la classe
    - @RequestMapping(value = "/lister" ...)
    - URL = classe + méthode ⇔ /comptes/lister
  - ▶ permet de définir
    - les variables d'URL : informations qui se trouvent sur l'URL
      - @RequestMapping(value = "/maj/{id}" ...)
    - le type de données en **entrées** et en **sorties**
      - @RequestMapping(produces = MediaType.APPLICATION\_JSON\_VALUE, consumes = ...)
    - les commandes HTTP supportées (get, put, ...)
      - @RequestMapping(method = { RequestMethod.GET, RequestMethod.PUT })

# Spring MVC : @RequestMapping

152

## ➤ Exemple :

```
105     @RequestMapping(value = "/byparam",
106                         method = { RequestMethod.GET, RequestMethod.PUT, RequestMethod.POST },
107                         produces = MediaType.APPLICATION_JSON_VALUE)
108     public @ResponseBody ResponseEntity<String> authentifierByParam(
109             @RequestParam(name = "login", required = true) String login,
110             @RequestParam(name = "password", required = true) String password) {
```

► Le web service sera joignable sur l'URL

- http(s)://[server]:[port]/[contextjee]/[urlmappingspringmvc]/[requestmappingclass]/[requestmappingmethode]
- http://localhost:8080/netbank/rest/authentifier/byparam?login=df&password=df

► Il fonctionnera en get, put et post, crachera du Json

# Spring MVC : @RequestParam

153

- Annotation sur les paramètres de la méthode
  - ... faireQQC(@RequestParam String login, ...)
- Permet de récupérer un paramètre provenant d'un formulaire ou d'un URL
- Le typage du paramètre n'a pas obligation d'être une chaîne de caractères
- Vous devez indiquer le nom du paramètre dans l'annotation
  - C'est le nom dans le formulaire ou la clé dans votre URL
  - Remarque : en Spring < 4, il n'y a pas d'attribut name pour cette annotation.
- Vous pouvez ajouter l'option *required* dans l'annotation pour indiquer le fait que le paramètre est obligatoire

# Spring MVC - @RequestParam

154

## ➤ Exemple :

```
105     @RequestMapping(value = "/byparam",
106                         method = { RequestMethod.GET, RequestMethod.PUT, RequestMethod.POST },
107                         produces = MediaType.APPLICATION_JSON_VALUE)
108     public ResponseEntity<String> authentifierByParam(
109             @RequestParam(name = "login", required = true) String login,
110             @RequestParam(name = "password", required = true) String password) {
```

- Login et password sont
  - des paramètres classiques
  - de type String
  - obligatoires

# Spring MVC : @RequestParam

- C'est Spring qui s'occupera de gérer la conversion si le typage n'est pas une String
- Si vous indiquez comme typage un objet à vous, et que le *consumes* indique JSON ou Xml, c'est aussi le Spring qui se chargera de faire la transformation
  - ❑ JAX-B pour l'XML
  - ❑ Jackson pour le Json où nom attribut = clef json
    - Attention aux majuscules / minuscules
- C'est aussi Spring qui va gérer les erreurs :
  - ❑ code 400 si le paramètre est absent (et en *required=true*)

# Spring MVC : @ResponseBody

- Annotation sur le retour de la méthode
  - ❑ public @ResponseBody ResponseEntity<String> faireQQC(...)
- Indique que le retour de la méthode représente le corps de la réponse
  - ❑ Peut être un objet de type ResponseEntity<T>
  - ❑ Peut être une String
  - ❑ Peut être un Object à vous (simple ex: Compte, ou composé List<Compte>)
- Dans le cas où Jackson est présent, c'est le Spring qui s'occupera de gérer les transformations Object -> Json

# Spring MVC : @ResponseBody

157

## ➤ Exemple :

```
105 @RequestMapping(value = "/byparam",
106                     method = { RequestMethod.GET, RequestMethod.PUT, RequestMethod.POST },
107                     produces = MediaType.APPLICATION_JSON_VALUE)
108     public @ResponseBody ResponseEntity<String> authentifierByParam(
109             @RequestParam(name = "login", required = true) String login,
110             @RequestParam(name = "password", required = true) String password) {
111 }
```

► Ici la réponse sera un objet ResponseEntity

- Un body
- Un HTTP Status

## ➤ Objet du Spring responsable de gérer

- ❑ Le contenu de la réponse : un Object transformé en JSON ou XML par vous ou le Spring
- ❑ Le code de retour : org.springframework.http.HttpStatus

## ➤ Rappels :

- ▶ [https://fr.wikipedia.org/wiki/Liste\\_des\\_codes\\_HTTP](https://fr.wikipedia.org/wiki/Liste_des_codes_HTTP)
- ▶ Selon qui à fait la requête HTTP (code Java, Javascript, client web, ...) si le code retour n'est pas 200 alors il faudra tester le comportement. Un client peut traiter le body d'une réponse 500 alors qu'un autre n'en fera rien

# JSON et org.springframework.http.ResponseEntity

159

- Exemple d'utilisation de ResponseEntity dans une méthode de WS

```
net.sf.json.JSONObject obj = new JSONObject();
obj.put("login", user.getLogin());
obj.put("nom", user.getNom());
obj.put("prenom", user.getPrenom());
obj.put("id", user.getId());
resu = new ResponseEntity<String>(obj.toString(), org.springframework.http.HttpStatus.OK);
```

- On fabrique un objet Json, ici via le framework SimpleJson
- On place dans la ResponseEntity le flux Json et un HttpStatus
  - ▶ Cet objet sera notre retour de méthode

# Spring MVC : @PathVariable

- Annotation sur les paramètres de la méthode
  - ... faireQQC(@PathVariable String smp, ...)
- Fonctionne avec **@RequestMapping** quand l'URL est composé d'informations dynamiques (avec des {})
  - @RequestMapping(value = "/maj/{smp}" ...)

# Spring mvc - @PathVariable

161

## ➤ Exemple :

```
61     @RequestMapping(value = "/byurl/{login}/{password}",  
62                         method = RequestMethod.GET,  
63                         produces = MediaType.APPLICATION_JSON_VALUE)  
64     public @ResponseBody ResponseEntity<String> authentifierByUrl(  
65                         @PathVariable String login,  
66                         @PathVariable String password) {
```

- Le web service sera joignable uniquement en GET sur l'URL
  - http(s)://[server]:[port]/[contextjee]/[urlmappingspringmvc]/[requestmappingclass]/[requestmappingmethode]/[var1]/[var2]
  - http://localhost:8080/netbank/rest/authentifier/byurl/df/df

# Spring mvc - @RequestBody

162

- @RequestBody est une annotation qui se place aussi sur les **paramètres** de la méthode
- Elle indique au Spring que l'élément est à récupérer du body de la requête. On l'utilise souvent dans le cas où le client envoie ses données en JSON ou XML.
- Le typage du paramètre
  - ➡ String : vous gérer le flux JSON vous-même dans votre méthode
  - ➡ Object : Spring va faire la transformation [Object↔ JSON] pour vous
- Attention : @RequestBody, sur un paramètre, **n'est pas utilisable avec un GET**
- Exemple :
  - ➡ La classe LoginBean sert vraiment de modèle au sens du MVC ↔ Objet d'échange entre la vue et le contrôleur.

```
58 @RequestMapping(method = {RequestMethod.POST},  
59           produces = MediaType.APPLICATION_JSON_VALUE,  
60           consumes = MediaType.APPLICATION_JSON_VALUE)  
61 public @ResponseBody ResponseEntity<String> authentifierPost(@RequestBody LoginBean loginBean) {
```

# Spring mvc - @ RequestBody

163

- La classe LoginBean
  - ➡ Objet *bête* ⇔ attribut + get/set
  - ➡ Le typage doit respecter les contraintes Json (pas de Date ou alors il faudra 's'arranger')
- Le flux Json (entrée ou sortie)

```
{  
    "login":"df",  
    "password":"df"  
}
```
- Attention : clef = getClef / setClef
  - ➡ Respectez scrupuleusement les majuscules minuscules

```
13 public class LoginBean implements Serializable {  
14  
15     private static final long serialVersionUID = 1L;  
16  
17     private String login;  
18     private String password;  
19  
20     /**  
21      * Constructeur de l'objet.  
22      */  
23     public LoginBean() {  
24         super();  
25     }  
26  
27     * Recupere la valeur du login..  
28     public String getLogin() {}  
29  
30     * Recupere la valeur du password..  
31     public String getPassword() {}  
32  
33     * Modifie la valeur du login..  
34     public void setLogin(String aValue) {}  
35  
36     * Modifie la valeur du password..  
37     public void setPassword(String aValue) {}  
38  
39     public String toString() {}  
40 }  
41 }
```

# Spring MVC : Session et Request

164

- En J2EE vous avez des objets standards
  - ▶ javax.servlet.http.HttpSession
  - ▶ javax.servlet.http.HttpServletRequest
  - ▶ javax.servlet.http.HttpServletResponse
- Si vous en avez besoin, ajoutez simplement comme paramètre un élément de type HttpSession, HttpServletRequest, HttpServletResponse
  - ▶ Le Spring se chargera de vous les donner
- Mais attention : vous n'êtes plus strictement en RESTFULL et cela peut compliquer vos codes de tests unitaires.

# Spring MVC : Gestion des erreurs

- Plutôt que de gérer les erreurs avec des try/catch dans vos méthodes, vous pouvez centraliser la gestion des erreurs dans
  - ▶ Tous vos contrôleurs à la fois
  - ▶ Chaque contrôleur indépendamment
- Avantages :
  - ▶ le code de vos contrôleurs est très simple à lire
  - ▶ Vous définissez le ResponseEntity en fonction de vos erreurs à un seul endroit

# Spring MVC : Gestion des erreurs

166

➤ Pour un contrôleur :  
annotation  
@org.springframework.  
work.web.bind.annotation.Exception  
Handler

```
1 package com.banque.web.controller.rest;
2
3+ import javax.servlet.http.HttpSession;[]
22
23 @RestController
24 @RequestMapping("/authentifier")
25 public class AuthentifierRestController2 {
26     private static final Logger LOG = LogManager.getLogger();
27-     @Autowired
28     private IAuthentificationService authenticationService;
29
30-     @RequestMapping(value = "/byurl/{login}/{pwd}", method = RequestMethod.GET, produces = MediaType.APPLICATION_JSON_VALUE)
31     @ResponseBody
32     public ResponseEntity<IUtilisateurEntity> authentifierByUrl(@PathVariable String login, @PathVariable String pwd,
33         HttpSession session) throws Exception {
34         AuthentifierRestController2.LOG.info("authentifier byurl RS login={} pwd=Xxxx", login);
35         if (login == null || pwd == null || login.trim().length() == 0 || pwd.trim().length() == 0) {
36             throw new Exception("Usage is /{login}/{password}");
37         }
38
39         IUtilisateurEntity user = this.authenticationService.authentifier(login, pwd);
40         if (session != null) {
41             session.setAttribute("user", user);
42         }
43         return new ResponseEntity<IUtilisateurEntity>(user, HttpStatus.OK);
44     }
45
46-     @ExceptionHandler(FonctionnelleException.class)
47     public ResponseEntity<FonctionnelleException> exceptionHandler(FonctionnelleException ex) {
48         ResponseEntity<FonctionnelleException> resu = new ResponseEntity<FonctionnelleException>(ex,
49             HttpStatus.BAD_REQUEST);
50         return resu;
51     }
52
53-     @ExceptionHandler(ErreurTechniqueException.class)
54     public ResponseEntity<ErreurTechniqueException> exceptionHandler(ErreurTechniqueException ex) {
55         ResponseEntity<ErreurTechniqueException> resu = new ResponseEntity<ErreurTechniqueException>(ex,
56             HttpStatus.INTERNAL_SERVER_ERROR);
57         return resu;
58     }
59 }
```

# Spring MVC : Gestion des erreurs

167

## ➤ Pour plusieurs contrôleurs : annotations

- @org.springframework.web.bind.annotation.ControllerAdvice
- Et @org.springframework.web.bind.annotation.ExceptionHandler

```
1 package com.banque.web.controller.rest;
2
3 import org.springframework.http.HttpStatus;
4 import org.springframework.http.ResponseEntity;
5 import org.springframework.web.bind.annotation.ControllerAdvice;
6 import org.springframework.web.bind.annotation.ExceptionHandler;
7
8 import com.banque.service.ex.ErreurTechniqueException;
9 import com.banque.service.ex.FonctionnelleException;
10
11 @ControllerAdvice
12 public class ExceptionControllerAdvice {
13
14     @ExceptionHandler(FonctionnelleException.class)
15     public ResponseEntity<FonctionnelleException> exceptionHandler(FonctionnelleException ex) {
16         ResponseEntity<FonctionnelleException> resu = new ResponseEntity<FonctionnelleException>(ex,
17             HttpStatus.BAD_REQUEST);
18         return resu;
19     }
20
21     @ExceptionHandler(ErreurTechniqueException.class)
22     public ResponseEntity<ErreurTechniqueException> exceptionHandler(ErreurTechniqueException ex) {
23         ResponseEntity<ErreurTechniqueException> resu = new ResponseEntity<ErreurTechniqueException>(ex,
24             HttpStatus.INTERNAL_SERVER_ERROR);
25         return resu;
26     }
27 }
```

# Spring MVC : Gestion des erreurs

- L'annotation `@ControllerAdvice` est paramétrable
  - ➡ **assignableTypes** : pour indiquer la liste des *class* visés (des classes annotées en `@Controller`)
  - ➡ **basePackages** : pour indiquer la liste des noms de packages contenant les `@Controller` visés
- Remarque : dans les exemples, en renvoie l'exception dans la `ResponseEntity` ⇔ c'est le framework Jackson qui se chargera de sa transformation en Json.
  - ➡ Cette transformation est brutale et rarement utilisable, il faudra compléter le code en spécifiant un flux Json pertinent pour vos applications

# Spring MVC : HATEOAS

- Hypermedia as the Engine of Application State
- De cette manière votre client REST n'a pas besoin de connaître les web services
  - ▶ Il peut les découvrir dynamiquement
- Le serveur peut changer ses web services sans impacter le client
  - ▶ A partir du moment où le client fait l'effort d'utiliser les liens (links/href)

# Spring MVC : HATEOAS

170

- Si vous souhaitez faire usage de cette norme vous pouvez l'intégrer dans vos projets Spring MVC REST via la dépendance à
  - org.springframework.hateoas / spring-hateoas
- Malheureusement, Spring ne va pas injecter automatiquement les links dans votre Json, vous devrez configurer vos objets Json afin de pointer vers les bonnes méthodes

# Spring MVC : HATEOAS

171

- Votre objet Json devra hériter de la classe  
`org.springframework.hateoas.ResourceSupport`
- Rappel : les id ne sont pas visibles dans les flux Json  
en HATEOAS

# Spring MVC : HATEOAS

172

## ➤ Exemple d'un objet Json paré pour HATEOAS

```
1 package com.banque.web.controller.rest.json;
2
3 import java.math.BigDecimal;
4
5 import org.springframework.hateoas.ResourceSupport;
6
7 import com.banque.entity.ICompteEntity;
8
9 public class CompteJson extends ResourceSupport {
10     private final ICompteEntity entity;
11
12     public CompteJson(ICompteEntity pEntity) {
13         super();
14         this.entity = pEntity;
15     }
16
17     public BigDecimal getTaux() {      return this.entity.getTaux();    }
18
19     public BigDecimal getDecouvert() {   return this.entity.getDecouvert(); }
20
21     public String getLibelle() {       return this.entity.getLibelle();    }
22
23     public BigDecimal getSolde() {     return this.entity.getSolde();     }
24
25     public Integer getUserId() {      return this.entity.getUserId();    }
26 }
```

# Spring MVC : HATEOAS

173

- Dans vos contrôleurs REST vous retournez des objets héritant **ResourceSupport**
- Pour chacun d'eux vous pourrez ajouter des attributs *links* via la méthode *add*
- Dans la méthode *add* vous ferez usage de la classe  
`org.springframework.hateoas.mvc.ControllerLinkBuilder` et de ses méthodes statiques  
`methodOn`, `linkTo`

```
75 @RequestMapping(value = "/{userId}/{cptId}", method = { RequestMethod.GET, RequestMethod.PUT,
76     RequestMethod.POST }, produces = MediaType.APPLICATION_JSON_VALUE)
77 @ResponseBody
78 public HttpEntity<CompteJson> voirCompte(@PathVariable(value = "userId") int userId,
79     @PathVariable(value = "cptId") int cptId) throws Exception {
80     ListerCompteRestController.LOG.info("voir un compte RS ({}) {}", String.valueOf(userId), String.valueOf(cptId));
81
82     ICompteEntity compte = this.compteService.select(userId, cptId);
83     CompteJson resu = new CompteJson(compte);
84     // Vers lui même
85     resu.add(ControllerLinkBuilder
86         .linkTo(ControllerLinkBuilder.methodOn(ListerCompteRestController.class).voirCompte(userId, cptId))
87         .withSelfRel());
88     // Vers l'utilisateur
89     resu.add(ControllerLinkBuilder
90         .linkTo(ControllerLinkBuilder.methodOn(AuthentifierRestController.class).voirUtilisateur(userId))
91         .withRel("utilisateur"));
92     return new ResponseEntity<CompteJson>(resu, HttpStatus.OK);
93 }
```

# Spring MVC : HATEOAS

174

## ➤ Résultat

```
[  
 {  
 "id":13,  
 "libelle":"Compte Courant",  
 "solde":1460.00,  
 "decouvert":100.00,  
 "taux":null,  
 "utilisateurId":2  
 },  
 {  
 "id":16,  
 "libelle":"Compte Remunéré",  
 "solde":540.00,  
 "decouvert":null,  
 "taux":0.30,  
 "utilisateurId":2  
 }  
 ]
```

Avant : Sans HATEOAS

```
[  
 {  
 "utilisateurId":2,  
 "solde":1460.00,  
 "decouvert":100.00,  
 "taux":null,  
 "libelle":"Compte Courant",  
 "links": [  
 {  
 "rel":"self",  
 "href":"http://localhost:8080/exo14.spring.mvc.rest.hateoas/rest/comptes/2/13"  
 },  
 {  
 "rel":"utilisateur",  
 "href":"http://localhost:8080/exo14.spring.mvc.rest.hateoas/rest/authentifier/2"  
 }  
 ],  
 {  
 "utilisateurId":2,  
 "solde":540.00,  
 "decouvert":null,  
 "taux":0.30,  
 "libelle":"Compte Remunéré",  
 "links": [  
 {  
 "rel":"self",  
 "href":"http://localhost:8080/exo14.spring.mvc.rest.hateoas/rest/comptes/2/16"  
 },  
 {  
 "rel":"utilisateur",  
 "href":"http://localhost:8080/exo14.spring.mvc.rest.hateoas/rest/authentifier/2"  
 }  
 ]  
 }  
 ]
```

Après : Avec HATEOAS

# Travaux pratiques

175

- En important l'exercice HATEOAS qui reprend les web services REST
  - ▶ La dépendance est déjà dans le Maven
  - ▶ Créez les classes de beans Json adaptés pour HATEOAS
  - ▶ Ajustez vos contrôleurs REST pour faire usage de HATEOAS



# Spring MVC : Junit – Avec MOCK

176

- On peut utiliser les Mock Spring pour tester nos WS sans les déployer sur un serveur J2EE
- Avec cette approche, il n'est plus nécessaire de déployer quoi que ce soit
  - ➡ Tests plus simple à écrire (pas de nom de serveur, ...)
  - ➡ Les tests seront lancés automatiquement avec nos tests unitaires

# Spring MVC : Junit – Avec MOCK

177

- Dans un premier temps il faut légèrement modifier la déclaration de nos tests Junit en complémentant les annotations :

```
6 package com.banque.test.rest;
7
8 import org.apache.logging.log4j.LogManager;
9 import org.apache.logging.log4j.Logger;
10 import org.junit.Assert;
11 import org.junit.Before;
12 import org.junit.Test;
13 import org.junit.runner.RunWith;
14
15 import org.springframework.beans.factory.annotation.Autowired;
16 import org.springframework.http.MediaType;
17 import org.springframework.test.context.ContextConfiguration;
18 import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
19 import org.springframework.test.context.web.WebAppConfiguration;
20 import org.springframework.test.web.servlet.MockMvc;
21 import org.springframework.test.web.servlet.ResultActions;
22 import org.springframework.test.web.servlet.request.MockMvcRequestBuilders;
23 import org.springframework.test.web.servlet.result.MockMvcResultMatchers;
24 import org.springframework.test.web.servlet.setup.MockMvcBuilders;
25 import org.springframework.web.context.WebApplicationContext;
26
27 import com.banque.web.model.LoginBean;
28 import com.fasterxml.jackson.databind.ObjectMapper;
29
30 /**
31 * Test sur la classe IAuthentificationService.
32 */
33 @RunWith(SpringJUnit4ClassRunner.class)
34 @ContextConfiguration(name = "applicationContext", locations = { "classpath:*-context.xml", "file:WebContent/WEB-INF/MVCDispatcher-servlet.xml" })
35 @WebAppConfiguration
36 public class TestUtilisateurWServicenD {
```

On Ajoute cette annotation

On complète sa liste de fichier Spring

# Spring MVC : Junit – Avec MOCK

178

- On ajoute deux attributs à notre test unitaire
  - Le premier de type WebApplicationContext qui sera en @Autowired. Cet objet va porter le contexte Spring.
  - Le second de type MockMvc. Cet objet va nous permettre de tester nos WS
- On ajoute aussi une méthode en @Before qui va initialiser notre objet MockMvc

```
42@  
43     @Autowired  
44     private WebApplicationContext webApplicationContext;  
45  
46  
48+     * Fabrication de notre mock MVC  
50@  
51     @Before  
52     public void setup() {  
53         this.mockMvc = MockMvcBuilders.webAppContextSetup(this.webApplicationContext).build();  
54     }
```

# Spring MVC : Junit – Avec MOCK

179

➤ Il ne reste plus qu'à réaliser la méthode de test

```
56+     * Usage des mock.■
57
58@    @Test
59    public void testAuthentifierOk() {
60        TestUtilisateurWSERVICE.LOG.trace("Test - testAuthentifierOk");
61        LoginBean loginBean = new LoginBean();
62        loginBean.setLogin("df");
63        loginBean.setPassword("df");
64        // Objet -> Json en Jackson
65        ObjectMapper mapper = new ObjectMapper();
66        try {
67            String loginBeanAsJson = mapper.writeValueAsString(loginBean);
68
69            // On appelle le web service en POST, en envoyant du JSON
70            // Attention : il n'y a plus d'URL complet (http://....)
71            // Juste l'URL du web service <-> @RequestMapping
72            ResultActions result = this.mockMvc.perform(MockMvcRequestBuilders.post("/authentifier")
73                    .contentType(MediaType.APPLICATION_JSON).content(loginBeanAsJson));
74            // On test que le resultat est bien 200
75            result.andExpect(MockMvcResultMatchers.status().isOk());
76            // Avec un flux JSON qui contient une proprietee id qui vaut 1
77            result.andExpect(MockMvcResultMatchers.content().contentType(MediaType.APPLICATION_JSON_VALUE))
78                    .andExpect(MockMvcResultMatchers.jsonPath("$.id").value(Integer.valueOf(1)));
79        } catch (Exception e) {
80            TestUtilisateurWSERVICE.LOG.error("Erreur", e);
81            Assert.fail(e.getMessage());
82        }
83    }
```

# Spring MVC : Junit – Avec MOCK

- La méthode perform de l'objet MockMvc permet de simuler l'envoie d'une requête HTTP.
- Elle prend en paramètre la commande HTTP
  - ➡ Get, post, put, ...
  - ➡ Et l'URL ⇔ ATTENTION : uniquement celui servant au mapping du WS ⇔ Celui indiqué dans l'annotation @RequestMapping
- On peut la compléter par des informations supplémentaires
  - ➡ Ici on précise que l'on va envoyer du Json et la chaîne de caractères (ligne 73)

# Spring MVC : Junit – Avec MOCK

181

- La méthode perform retourne un objet **ResultActions**
- Sur cet objet vous pouvez appeler d'autres méthodes de validations via l'appel à andExpect()
- Si vous voulez récupérer le status ou la réponse, vous devrez passer par la méthode andReturn() qui vous donnera un objet **MvcResult**
  - Il suffit ensuite d'appeler par exemple getResponse().getContentAsString() pour récupérer son flux Json de réponse
  - Ou getResponse().getStatus() pour récupérer le HttpStatus code
  - Ou getResponse().getRequest().getSession() pour récupérer une HttpSession

# Spring MVC : Junit – Avec MOCK

182

- Dans le cas où vous avez fait usage de l'objet HttpSession, le code devient un peu plus complexe
- Méthode andReturn().getRequest().getSession() sur notre objet ResultActions
- 1er appel REST

```
// On appelle le web service en POST, en envoyant du JSON
// Attention : il n'y a plus d'URL complet (http://....)
// Juste l'URL du web service <=> @RequestMapping
ResultActions result = this.mockMvc.perform(MockMvcRequestBuilders.post("/authentifier")
    .contentType(MediaType.APPLICATION_JSON).content(loginBeanAsJSON));
```

- Récupération de la session
- 2eme appel avec la session

```
// On récupère la session pour la repasser à la requête suivante
// On fait cela car notre WS se sert de HttpSession
session = result.andReturn().getRequest().getSession();
```

```
// On appelle le web service en GET, en remettant la session
ResultActions result = this.mockMvc
    .perform(MockMvcRequestBuilders.get("/comptes").session((MockHttpSession) session));
```

# Spring MVC : Junit – Avec MOCK

183

```
93+     * Usage des mock.□
95@     @Test
96     public void testListerCompteOk() {
97         TestUtilisateurWSERVICE.LOG.trace("Test - testListerCompteOk");
98         // Bean qui sert à l'authentification
99         LoginBean loginBean = new LoginBean();
100        loginBean.setLogin("df");
101        loginBean.setPassword("df");
102        // Objet session qui permet de propager la session dans nos requetes qui
103        // suivent l'authentification
104        HttpSession session = null;
105        UtilisateurEntity objRetour = null;
106        // Objet -> Json en Jackson
107        ObjectMapper mapper = new ObjectMapper();
108        try {
109            String loginBeanAsJson = mapper.writeValueAsString(loginBean);
110
111            // On appelle le web service en POST, en envoyant du JSON
112            // Attention : il n'y a plus d'URL complet (http://....)
113            // Juste l'URL du web service <=> @RequestMapping
114            ResultActions result = this.mockMvc.perform(MockMvcRequestBuilders.post("/authentifier")
115                .contentType(MediaType.APPLICATION_JSON).content(loginBeanAsJson));
116            // On test que le resultat est bien 200
117            result.andExpect(MockMvcResultMatchers.status().isOk());
118            // Avec un flux JSON qui contient une proprietee id qui vaut 1
119            result.andExpect(MockMvcResultMatchers.content().contentType(MediaType.APPLICATION_JSON_VALUE))
120                .andExpect(MockMvcResultMatchers.jsonPath("$.id").value(Integer.valueOf(1)));
121            // On recupere la session pour la repasser a la requette suivante
122            // On fait cela car notre WS se sert de HttpSession
123            session = result.andReturn().getRequest().getSession();
124            // Recuperation du flux en JSON
125            String fullStringResult = result.andReturn().getResponse().getContentAsString();
126            TestUtilisateurWSERVICE.LOG.debug("JSON Result is {}", fullStringResult);
127            // Json -> Object en Jackson
128            objRetour = mapper.readValue(fullStringResult, UtilisateurEntity.class);
129        } catch (Exception e) {
130            TestUtilisateurWSERVICE.LOG.error("Erreur", e);
131            Assert.fail(e.getMessage());
132        }
```

# Spring MVC : Junit – Avec MOCK

184

```
133     Assert.assertNotNull("La session ne doit pas être null", session);
134     Assert.assertNotNull("La session doit contenir le user id", session.getAttribute("userId"));
135     Assert.assertNotNull("L'objet traduit du Json ne doit pas etre null", objRetour);
136     Assert.assertEquals("Le user id est le même que celui du flux JSON", objRetour.getId(),
137                         session.getAttribute("userId"));
138
139     // On va maintenant recuperer les comptes de l'utilisateur connecté
140     try {
141         // On appelle le web service en GET, en remettant la session
142         ResultActions result = this.mockMvc
143             .perform(MockMvcRequestBuilders.get("/comptes").session((MockHttpSession) session));
144         // On test que le résultat est bien 200
145         result.andExpect(MockMvcResultMatchers.status().isOk());
146         // Avec un flux JSON qui contient une propriété liste qui n'est pas
147         // null
148         result.andExpect(MockMvcResultMatchers.content().contentType(MediaType.APPLICATION_JSON_VALUE))
149             .andExpect(MockMvcResultMatchers.jsonPath("$.liste").value(CoreMatchers.notNullValue()));
150         // Recupération du flux en JSON
151         String fullStringResult = result.andReturn().getResponse().getContentAsString();
152         TestUtilisateurWSERVICEND.LOG.debug("JSON Result is {}", fullStringResult);
153     } catch (Exception e) {
154         TestUtilisateurWSERVICEND.LOG.error("Erreur", e);
155         Assert.fail(e.getMessage());
156     }
```

# Spring MVC : Junit – Avec RestTemplate

185

- Pour tester vos contrôleurs WS il vous faut simuler/créer des requêtes HTTP
- En Spring, l'objet `org.springframework.web.client.RestTemplate` vous permettra de le faire très simplement
- Il suffit d'instancier un `RestTemplate` au moment où vous souhaitez envoyer votre requête
- En fonction des méthodes du `RestTemplate` que vous utiliserez cela enverra
  - ▶ un GET, PUT, POST, ...
  - ▶ avec des paramètres
  - ▶ avec du JSON / XML
  - ▶ ...

# Spring MVC : Junit – Avec RestTemplate

186

- Point important : pour fonctionner l'objet RestTemplate a besoin que votre web service soit déployé sur un serveur

## ➤ Méthodes du RestTemplate

- ▶ Premier paramètre : URL
- ▶ Second paramètre de type Class : le type de retour
- ▶ Second paramètre de type Objet : l'élément à envoyer
- ▶ Dernier paramètre ... : Pour boucher les trous dans l'URL

HTTP	RESTTEMPLATE
DELETE	delete(String, String...)
GET	getForObject(String, Class, String...)
HEAD	headForHeaders(String, String...)
OPTIONS	optionsForAllow(String, String...)
POST	postForLocation(String, Object, String...)
PUT	put(String, Object, String...)

# Spring MVC : Junit – Avec RestTemplate

187

- Exemple d'utilisation (GET) : envoyer des paramètres => récupérer un flux JSON sous forme de String

```
27+     * Test..
28-
29@Test
30 public void testAuthentifierOk() {
31     TestUtilisateurWSERVICE.LOG.trace("Test - testAuthentifierOk");
32     final String login = "df";
33     final String pwd = "df";
34     RestTemplate restTemplate = new RestTemplate();
35     String resu = null;
36     try {
37         resu = restTemplate.getForObject(
38             "http://localhost:8080/netbank/rest/authentifier/byparam?login={login}&password={pwd}",
39             String.class, login, pwd);
40     } catch (RestClientException e) {
41         TestUtilisateurWSERVICE.LOG.error("Erreur", e);
42         Assert.fail(e.getMessage());
43     }
44     TestUtilisateurWSERVICE.LOG.debug("Test - testAuthentifierOk - {}", resu);
45 }
```

# Spring MVC : Junit – Avec RestTemplate

188

- Exemple d'utilisation (POST) : envoyer un flux JSON => Récupérer un flux
  - ➡ Dans cet exemple, le framework Jackson est une dépendance du projet

```
42 @Test
43 public void testAuthentifierOk() {
44     TestUtilisateurWSERVICE.LOG.trace("Test - testAuthentifierOk");
45     RestTemplate restTemplate = new RestTemplate();
46     LoginBean loginBean = new LoginBean();
47     loginBean.setLogin("df");
48     loginBean.setPassword("df");
49     try {
50         // Envoie de login/pwd en JSON en post
51         restTemplate.postForLocation(TestUtilisateurWSERVICE.AUTHENTIFIER_URL, loginBean);
52     } catch (RestClientException e) {
53         TestUtilisateurWSERVICE.LOG.error("Erreur", e);
54         Assert.fail(e.getMessage());
55     }
56 }
```

# Spring MVC : Junit – Avec RestTemplate

189

- Vous pouvez récupérez la réponse dans son intégralité en passant par la méthode `exchange`
- Dans l'exemple qui suit :
  - ▶ On envoie une première requête HTTP en POST avec un objet JSON
  - ▶ A travers la réponse, on récupère un JSESSIONID
    - On fait cela car côté serveur dans cet exemple, le WS fait usage de l'objet HttpSession et on en a besoin pour faire les autres appels (problématique de l'authentification)
  - ▶ Dans la seconde requête on demande en GET la liste des comptes de l'utilisateur connecté
  - ▶ Dans les deux cas, on obtient un flux JSON

# Spring MVC : Junit – Avec RestTemplate

190

```
91     @Test
92     public void testListerCompteOk2() {
93         RestTemplate restTemplate = new RestTemplate();
94         // Notre bean/model pour le JSON en envoie
95         LoginBean loginBean = new LoginBean();
96         loginBean.setLogin("df");
97         loginBean.setPassword("df");
98         // Indiquons que l'on accept le Json
99         HttpHeaders headers = new HttpHeaders();
100        headers.setAccept(Arrays.asList(MediaType.APPLICATION_JSON));
101        HttpEntity<?> entity = new HttpEntity<>(loginBean, headers);
102
103        ResponseEntity<String> result = restTemplate.exchange(TestUtilisateurWSService.AUTHENTIFIER_URL, HttpMethod.POST,
104                    entity, String.class);
105        TestUtilisateurWSService.LOG.debug("Resultat-status {}", result.getStatusCode());
106        TestUtilisateurWSService.LOG.debug("Resultat-flux de retour {}", result.getBody());
107
108        // Afin de lister le contenu du header de la réponse
109        // for (Map.Entry<String, List<String>> elm :
110        // result.getHeaders().entrySet()) {
111        // TestUtilisateurWSService.LOG.debug("{}={}", elm.getKey(),
112        // elm.getValue());
113        // }
114
115        // Pour la seconde requete on recuper le JSESSIONID et on le repousse
116        // On fait cela car notre WS se sert de HttpSession
117        headers.clear();
118        headers.add("Cookie", result.getHeaders().get("Set-Cookie").get(0));
119        headers.setAccept(Arrays.asList(MediaType.APPLICATION_JSON));
120        entity = new HttpEntity<?>(null, headers);
121        result = restTemplate.exchange(TestUtilisateurWSService.LISTER_CPT_URL, HttpMethod.GET, entity, String.class);
122        TestUtilisateurWSService.LOG.debug("Resultat-status {}", result.getStatusCode());
123        TestUtilisateurWSService.LOG.debug("Resultat-flux de retour {}", result.getBody());
124    }
```

# Spring et Ajax

- Rien de particulier sur ce point, le Spring ne vous aidera pas à faire de l'Ajax, vous utiliserez vos librairies Java Script habituelles :
  - ▶ JQuerry
  - ▶ Angular
  - ▶ ...
- Vous ferez par contre fortement usage du Json et donc du @ResponseBody et @RequestBody
  - ▶ Et certainement du framework Jackson

# Travaux pratiques

192

- Modifions le site web afin de le compléter avec des webs services REST

