Spring
REST

Objectifs du chapitre

- Dans ce chapitre nous allons
  - utiliser Ajax
  - mettre en place des URL REST

REST

- REpresentational State Transfert
  - architecture créée en 2000 par Roy Fielding
- Architecture
  - client-serveur, sans état
  - interface uniforme, 4 règles
    - chaque ressource est identifiée de manière unique (URI)
    - les ressources ont des représentations définies
      - les méta-données permettent au client de modifier l'état de la ressource
    - message auto-descriptif
    - moteur d'état hypermédia

RESTFul

- Le marketing met en avant les web services de type REST et RESTful
- Les web services RESTful utilisent de manière explicite les méthodes HTTP
  - GET pour récupérer une ressource
  - POST pour créer une ressource
  - PUT pour modifier une ressource
  - DELETE pour supprimer une ressource

RESTful

- Des en-têtes de requêtes HTTP plus explicites
  - avec une application web classique

```
GET /adduser?name=Toto HTTP/1.1
```

  - avec RESTful

```
POST /users HTTP/1.1
Host : localhost
Content-Type: application/xml
<?xml version="1.0"?>
<user>
    <name>Toto</name>
</user>
```

RESTful

- Des URIs plus propres
  - plus intuitive

```
http://my.domain.org/discussion/topics/java
```

```
http://my.domain.org/discussion/2011/12/23/java
```

  - qui peuvent être analysées

```
http://my.domain.org/discussion/topics/{topic}
```

```
http://my.domain.org/discussion/{year}/{month}/{day}/{topic}
```

RESTful

- Règles pour la structure d'une URI d'un web service RESTful
  - cacher la technologie utilisée par le serveur
    - pas d'extension .jsp, .php, etc.
  - tout en minuscule
  - les espaces sont remplacés par des - ou _
  - évite les requêtes SQL dans les URL
  - toujours fournir une page par défaut
    - à l'instar du code 404 Not Found

RESTful

- Les transferts s'effectuent généralement en
  - XML - POX (Plain Old XML)
    - type MIME : application/xml
  - JSON (JavaScript Object Notation)
    - type MIME : application/json
  - XHTML
    - type MIME : application/xhtml+xml
- D'autres format existent
  - texte pur, YAML, ...

RESTful

- Exemples POX et JSON

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<identite>
    <civilite>M</civilite>
    <prenom>Gaston</prenom>
    <nom>LAGAFFE</nom>
    <adresse>
        <rue>Rue de Bruxelles</rue>
        <ville>Paris</ville>
        <code-postal>75000</code-postal>
    </adresse>
</identite>
```

```json
{
    "civilite":"M",
    "prenom":"Gaston",
    "nom":"LAGAFFE",
    "adresse":{
        "rue":"Rue de Bruxelles",
        "ville":"Paris",
        "codePostal":"75000"
    }
}
```

JAXB - Sérialisation XML et Java

- Le passage Java ↔ XML est grandement facilité par l'API JAXB
  - Java Architecture for XML Binding
  - facilite la manipulation des documents XML
    - avec JAXP (SAX et DOM) le traitement des données XML est à coder
  - analyse le schema XML
  - génère un ensemble de classes

JAXB - Sérialisation XML et Java

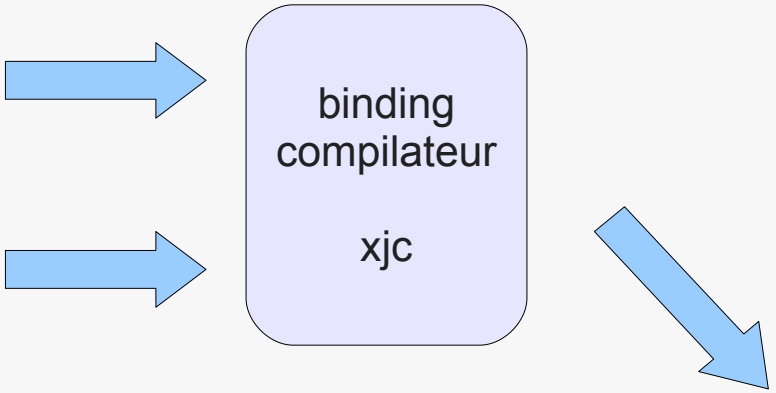
- Deux étapes principales
  - génération des classes et interfaces à partir du schéma XML
  - utilisation des classes générées pour transformer un document XML en graphe d'objets, ou inversement
  - Spring rend transparent ces étapes

Utilisation de JAXB

XML Schema

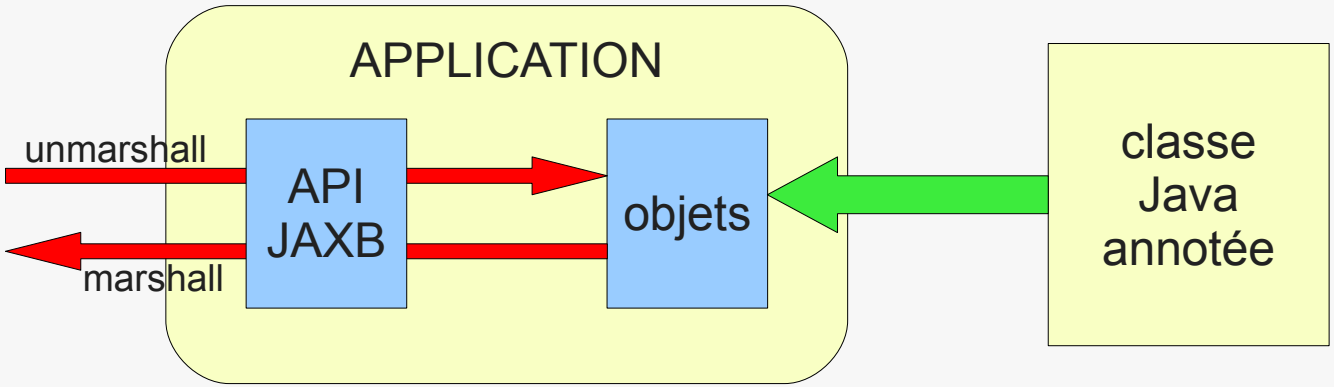
```
<xs:element name="entreprise">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="raison_sociale" type="xs:string"/>
            <xs:element ref="adresse"/>
            <xs:element name="web"/>
        </xs:sequence>
        <xs:attribute name="id" type="xs:ID" use="required"/>
    </xs:complexType>
</xs:element>
```

configuration



document XML

```
<entreprise id="e2">
    <raison_sociale>Le bateau ivre</raison_sociale>
    <adresse type="pro">
        <rue>3, rue des Vagues</rue>
        <code_postal>35000</code_postal>
        <ville>RENNES</ville>
        <pays>FRANCE</pays>
    </adresse>
    <web>www.le-bateau-ivre.bzh</web>
</entreprise>
```



JAXB

- Les classes peuvent être annotées pour pouvoir être sérialisée dans un fichier XML
  - une classe doit être `@XmlRootElement`
  - cf. documentation pour ensemble des annotations
- Un fichier de configuration peut être ajouté

```
<?xml version="1.0" encoding="UTF-8"?>
<bindings xmlns="http://java.sun.com/xml/ns/jaxb"
    xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/jaxb
                http://java.sun.com/xml/ns/jaxb/bindingschema_2_0.xsd"
    version="2.1">
        <schemaBindings>
            <package name="org.antislashn.jaxb" />
        </schemaBindings>
</bindings>
```

RESTful avec Spring

- Structure des URI utilisées

```
http://localhost:8080/nom_appli/contacts/{id}
```

  - où `{id}` est le paramètre correspondant à l'identifiant
    - exemple

```
http://localhost:8080/nom_appli/contacts/1
```

  - en fonction de la méthode HTTP le comportement du web service sera différent

RESTful avec Spring

- Utilisation de XML comme format d'échange, via l'API JAXB
  - ajouter la dépendance Maven

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

  - Spring fournit des beans qui pourront prendre en charge le marchalling et unmarshalling de nos classes

RESTful avec Spring

- Classiquement les requêtes arrivent sur un `Controller`
  - détails de l'annotation `@RequestMapping` et exemples plus loin
- La réponse pourrait-être générée "manuellement" par une JSP
- Spring propose une solution alternative plus simple
  - le contrôleur ne renverra pas une chaîne de caractères correspondant à une vue, mais à un bean

RESTful avec Spring

- Les services sont exposés par l'annotation `@RequestMapping`
  - `String[] consumes` : types MIME liés à la requête
  - `String[] headers` : en-têtes liés à la requête
  - `String[] method` : méthodes HTTP de la requête
  - `String[] produces` : types MIME produits
  - `String[] params` : paramètres de la requête
  - `String[] value` : mapping de base de la requête

RESTful avec Spring

- Les paramètres son automatiquement récupérés par les annotations
  - `@PathVariable`
    - permet de lier un paramètre de méthode avec le modèle de l'URI
  - `@RequestBody`
    - permet de lier le corps d'une requête HTTP (POST, PUT) avec un paramètre de méthode

RESTful avec Spring

- Exemple
  - méthode GET


```
@Controller
@RequestMapping(value="/contacts")
public class ContactController {

@RequestMapping(value="/all",method=RequestMethod.GET)
public String getAllContacts(Model model)
...
```

RESTful avec Spring

- Exemple
  - méthode GET


```
@Controller
@RequestMapping(value="/contacts")
public class ContactController {

@RequestMapping(value="/id/{id}",method=RequestMethod.GET)
public String getContactById(@PathVariable("id") int idContact, Model model){
...
```

pas nécessaire si le nom PathVariable est le même que le nom du paramètre

RESTful avec Spring

- Exemple
  - méthode POST
    - une instance de contact est sérialisée dans le corps de la requête HTTP


```
@RequestMapping(value="/contact",method={RequestMethod.POST,RequestMethod.PUT})
public String addOrUpdate(@RequestBody Contact contact,Model model){
...
}
```

RESTful avec Spring
Renvoyer du XML

- Il est nécessaire d'annoter les classes du domaine à transformer en XML avec `@XmlRootElement`

```
@XmlRootElement(name="contact")
public class Contact implements Serializable{
...
}
```

- Les collections doivent aussi être annotées

```
@XmlRootElement(name="contacts")
public class ContactList {

    private List<Contact> contacts = new ArrayList<Contact>();

    @XmlElement(name="contact")
    public List<Contact> getContactsList(){
        return contacts;
    }
...
}
```

RESTful avec Spring
Renvoyer du XML

- Le fichier *servlet-context.xml* est modifié pour :
  - ajouter une résolution des vues par bean

```
<beans:bean class="org.springframework.web.servlet.view.BeanNameViewResolver" />
```

  - le marshalling des POJO

```
<beans:bean id="contactXmlTemplate"
                    class="org.springframework.web.servlet.view.xml.MarshallingView">
    <beans:constructor-arg>
        <beans:bean class="org.springframework.oxm.jaxb.Jaxb2Marshaller">
            <beans:property name="classesToBeBound">
                <beans:list>
                    <beans:value>org.antislashn.contacts.domain.Contact</beans:value>
                    <beans:value>org.antislashn.contacts.domain.ContactList</beans:value>
                </beans:list>
            </beans:property>
        </beans:bean>
    </beans:constructor-arg>
</beans:bean>
```

RESTful avec Spring
Renvoyer du XML

- Le contrôleur renvoie une valeur correspondant au nom du bean `MarshallingView`

```
@RequestMapping(value="/all",method=RequestMethod.GET)
public String getAllContacts(Model model){
     model.addAttribute("contacts",service.getAllContacts());
     return "contactXmlTemplate";
}

@RequestMapping(value="/id/{id}",method=RequestMethod.GET)
public String getContactById(@PathVariable("id") int idContact, Model model){
     model.addAttribute("contact",service.getContactById(idContact));
     return "contactXmlTemplate";
}
```

RESTful avec Spring
Renvoyer du XML

- Annotation `@ResponseBody`
  - si la méthode du contrôleur renvoie directement un objet et qu'il existe une sérialisation pour cet objet, Spring renvoie l'objet sous sa forme sérialisée
    - utilisation d'un `HttpMessageConverter`

```
@RequestMapping(value="/id/{id}",method=RequestMethod.GET)
@ResponseBody
public Contact getContactById(@PathVariable("id") int idContact){
     return service.getContactById(idContact);
}
```

    - il n'est plus nécessaire d'avoir le bean `MarshallingView`

Consommation d'un RESTful XML

- Un simple projet Spring suffit
  - il faut ajouter les dépendances Maven suivantes :
    - spring-web
    - spring-oxm

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

Consommation d'un RESTful XML

- Afin d'activer la sérialisation / dé-sérialisation des objets
  - ajoutez dans le fichier de configuration Spring

```
<bean id="xmlMarshaller" class="org.springframework.oxm.jaxb.Jaxb2Marshaller">
     <property name="classesToBeBound">
          <list>
               <value>org.antislashn.contacts.domain.Contact</value>
               <value>org.antislashn.contacts.domain.ContactList</value>
          </list>
     </property>
</bean>

<bean id="restTemplate" class="org.springframework.web.client.RestTemplate">
     <property name="messageConverters">
          <list>
               <bean class="org.springframework.http.converter.xml.MarshallingHttpMessageConverter">
                    <property name="marshaller" ref="xmlMarshaller" />
                    <property name="unmarshaller" ref="xmlMarshaller" />
               </bean>
          </list>
     </property>
</bean>
```

Consommation d'un RESTful XML

- L'instance de `RestTemplate` sera utilisée pour envoyer les requêtes vers le serveur
  - méthodes :
    - `getForObject()`
    - `put()`
    - `delete()`
    - `postForObject()`
    - ...

Consommation d'un RESTful XML

- Exemples

```
private String baseUrl = "http://localhost:8080/restful/contacts/";
```

  - récupérer un contact par son identifiant

```
public Contact getContactById(int id){
     String url = baseUrl + "id/"+id;
     Contact c = restTemplate.getForObject(url, Contact.class);
     return c;
}
```

  - ajouter ou changer un contact

```
public void add(Contact contact){
     String url = baseUrl + "contact";
     restTemplate.put(url, contact);
}
```

RESTful avec Spring
Renvoyer du JSON

- Il est nécessaire d'ajouter une dépendance Maven

```
<dependency>
     <groupId>org.codehaus.jackson</groupId>
     <artifactId>jackson-mapper-asl</artifactId>
     <version>[1.9,)</version>
</dependency>
```

- Les méthodes renvoient directement l'objet
  - celui-ci est converti en JSON
    - s'il n'est pas annoté comme étant du XML

RESTful avec Spring

- Si plusieurs formats doivent être gérés pour créer la réponse
  - XML, JSON, PDF, …
- Stratégies possibles
  - une URI distinct par type de format
  - utiliser le type MIME accepté par le client et faire de la négociation de contenu
    - variable *Accept* de l'en-tête HTTP, mais les clients utilisent plusieurs types MIME