MODULE Spring
Spring REST

HUMAN booster
VOTRE SOLUTION COMPETENCE

Spring REST

Plan du module Spring REST

- Qu’est-ce que REST ?
- REST avec Spring
- Les contrôleurs REST
- HATEOAS avec Spring

Spring REST

Qu’est-ce que REST ?

Qu’est-ce que REST ?

(Présentation REST non incluse)

Spring REST

REST avec Spring

REST avec Spring

Spring MVC vise à faciliter la création d'applications dans lesquelles un humain intervient :

- pour prendre des actions, comme cliquer sur un bouton de lien, saisir des données dans un formulaire, etc.
- Pour visualiser des résultats.

Autrement dit, on parle principalement d'applications de gestion de sites Internet, intranet, ou autre application avec IHM.

REST avec Spring

Le style d’architecture REST peut se mettre en œuvre pour réaliser des applications avec IHM ou sans IHM, autrement dit des applications qui ne nécessitent pas d’intervention humaine.

Cela implique en particulier que les réponses du serveur ne seront **pas formattées en HTML**, comme avec une application MVC.

Ce qui est retourné par le serveur peut être n’importe quoi : du texte ASCII, du binaire (image, vidéo, etc.), du XML, du JSON, etc.

REST avec Spring

Autre différence fondamentale avec une application Spring MVC : les **codes de retour HTTP**.

Dans une telle application, les codes de retour ne revêtent pas une importance très grande, car on fait confiance à un humain pour comprendre ce qui est affiché par le navigateur.

Quand ce sont deux applications informatiques qui communiquent, le seul moyen pour qu'elles se comprennent est qu'elles partagent un **langage commun**.

REST avec Spring

Ce langage commun consiste d'une part en des codes de retour **standard** et d'autre part en des requêtes **standard**.

D'où la 3ème différence fondamentale : en REST, les verbes HTTP ont une **signification bien précise**. A la différence d'une application gérant un site Internet, ou d'une application SOAP ou XML-RPC, dans lesquelles on utilise indifféremment GET ou POST, et quasiment jamais les autres verbes.

REST avec Spring

La plupart des services et utilitaires Spring pour REST sont disponibles via la dépendance Spring Web (celle utilisée pour Spring MVC) :

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

REST avec Spring

Les services de Spring pour HATEOAS sont disponibles via la dépendance suivante :

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

Spring REST

Les contrôleurs REST

Les contrôleurs REST

On a vu que Spring MVC définit le concept de **contrôleur** comme support pour grouper les traitements côté serveur.

Spring propose le concept de « contrôleur REST » pour supporter les traitements REST côté serveur.

La grande différence d'avec un contrôleur MVC, c'est que les données de sortie ne seront **pas générées par une vue**. Normal, une vue génère du HTML, qui est quelque chose de visualisable par un humain au travers d'un navigateur Web...

Les contrôleurs REST

Les données de sortie seront donc attachées à la réponse de la servlet (`HttpServletResponse`), sans passage par un résolveur de vue puis une vue.

Spring fournit la classe générique `org.springframework.http.ResponseEntity` pour représenter une réponse manipulable simplement.

De nombreux utilitaires Spring permettent de gérer simplement les réponses.

Les contrôleurs REST

Le code de retour HTTP sera choisi **précisément** et ajouté à cette réponse.

Les headers HTTP doivent aussi être ajoutés à la réponse, en particulier :

- Le type MIME de la réponse (`Content-Type`)
- La localisation (`Location`)
- L'Etag (`ETag`)
- La date d'expiration (`Expires`)
- ...

Les contrôleurs REST

Bien évidemment, Spring permet de faire cela de plein de manières différentes et fournit beaucoup d'utilitaires pour simplifier ces opérations.

Exemple : positionner toujours le même code de retour pour un traitement donné

```
@ResponseStatus(code = HttpStatus.CREATED)

ResponseEntity<MaClasse> ajouteInstance(…) { … }
```

Les contrôleurs REST

Autre exemple : indiquer à Spring que la valeur retournée par n'importe quelle méthode d'un contrôleur doit toujours être utilisée pour construire le "body" de la réponse HTTP.

```
@RestController

class MonControleurREST { … }
```

Cette indication peut aussi être portée par une méthode de contrôleur au moyen de l'annotation `@ResponseBody`.

Les contrôleurs REST

@ResponseBody

```
@ResponseStatus(code = HttpStatus.CREATED)

ResponseEntity<MaClasse> ajouteInstance(…) { … }
```

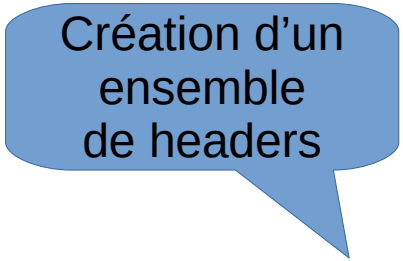
Les contrôleurs REST

Pour construire les headers, Spring propose la classe `org.springframework.http.HttpHeaders`, qui est une Map proposant :

- une méthode générique pour ajouter un header, en spécifiant le nom du header et sa valeur
- diverses méthodes dédiées chacune à un header précis (`Accept-*`, `Content-Length`, `Content-Type`, `Location`, `Etag`, `Expires`, `Origin`, `Date`, `Last-Modified`, etc.)

Les contrôleurs REST

Exemple.




```
URI location = ...;

HttpHeaders responseHeaders = new HttpHeaders();

responseHeaders.setLocation(location);

responseHeaders.set("MyResponseHeader", "MyValue");

return new ResponseEntity<String>("Hello World",
   responseHeaders, HttpStatus.CREATED);
```

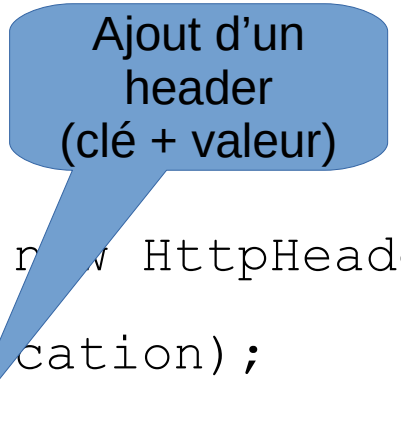
Les contrôleurs REST

Exemple.


```
URI location = ...;

HttpHeaders responseHeaders = new HttpHeaders();

responseHeaders.setLocation(location);

responseHeaders.set("MyResponseHeader", "MyValue");

return new ResponseEntity<String>("Hello World",
   responseHeaders, HttpStatus.CREATED);
```

Les contrôleurs REST

Exemple.




```
URI location = ...;

HttpHeaders responseHeaders = new HttpHeaders();

responseHeaders.setLocation(location);

responseHeaders.set("MyResponseHeader", "MyValue");

return new ResponseEntity<String>("Hello World",
   responseHeaders, HttpStatus.CREATED);
```

Les contrôleurs REST

Exemple.

```
URI location = ...;

HttpHeaders responseHeaders = n    ttpHeaders();

responseHeaders.setLocation(l      ation);

responseHeaders.set("MyRes      onseHeader", "MyValue");

return new ResponseEntity<String>("Hello World",
   responseHeaders, HttpStatus.CREATED);
```

Création d'une réponse HTTP avec body, headers et code de retour

Les contrôleurs REST

Autre exemple de création d'une réponse complète.

```
ResponseEntity<Employe> nouvelEmploye(@RequestBody Employe
nouvelEmploye, HttpServletRequest requete) {

  Employe employe = repository.save(newEmployee);

  String urlRequete = requete.getRequestURL().toString();

  URI location = new URI(urlRequete + "/" + employe.getId());

  return
ResponseEntity.created(location).header("MyResponseHeader",
"MyValue").body(employe);

}
```

Spring REST

HATEOAS avec Spring

HATEOAS avec Spring

Les services de Spring pour HATEOAS sont disponibles via la dépendance suivante :

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

HATEOAS avec Spring

HATEOAS est un moyen de découpler le client et le serveur, le dernier founissant des liens pour activer les actions sur les ressources qu'il gère, afin que le premier n'ait pas besoin de connaître précisément la logique de mainpulation de ces ressources.

Exemple : après création d'une ressource, le serveur peut retourner dans sa réponse un lien pour accéder à la ressource, pour la modifier ou pour la supprimer.

HATEOAS avec Spring

Autre exemple.

Si une opération complexe nécessite plusieurs étapes (comme l'achat de biens sur Internet), il y aura oplusieurs échanges entre le client et le serveur.

Lorsque la dernière étape est atteinte, le serveur peut retourner deux liens :

- l'un pour confirmer la transaction
- l'autre pour annuler la transaction.

HATEOAS avec Spring

Dans cette manière de faire, c'est **le serveur** qui sait quand la transaction peut être confirmée ou annuler, pas le client. Ce dernier n'a qu'à proposer les liens sur le naviogateur, avec le texte envoyé par le serveur. Le **couplage** entre le serveur et le client est donc **réduit**.

Une modification de la logique de l'opération (ajout d'une étape, par exemple pour valider un bon de réduction) serait faite **uniquement** du côté du serveur, ce qui simplifie énormément les évolutions du système.

HATEOAS avec Spring

HATEOAS consiste à enrichir une réponse pour y ajouter des liens pour réaliser d'autres opérations, qui sont valides dans l'état atteint par la ressource au moment où ces liens sont définis.

Spring fournit des services pour définir ces liens, en particulier la classe

`org.springframework.hateoas.server.mvc.WebMvcLinkBuilder`.

Exemple de création de liens page suivante.

HATEOAS avec Spring

```
@GetMapping("/employees/{id}")
EntityModel<Employe> findOne(@PathVariable Long id) {
    Employe employe = repository.findById(id)
      .orElseThrow(() -> new EmployeNotFoundException(id));
    EntityModel<Employe> modele = new EntityModel<>(employe);
    Link link = WebMvcLinkBuilder.linkTo(EmployeeController.class).withRel("root");
    modele.add(link);
```

HATEOAS avec Spring

```
    link = WebMvcLinkBuilder.linkTo(WebMvcLinkBuilder.methodOn(EmployeeController.class).findOne(id)).withSelfRel();

    modele.add(link);

    link = WebMvcLinkBuilder.linkTo(WebMvcLinkBuilder.methodOn(EmployeeController.class).all()).withRel("all");

    modele.add(link);

    return modele;

}
```

HATEOAS avec Spring

Mise en pratique :

- Exercice 01 : une application Spring REST basique