Ingénierie Web

INSA Rennes -- 2022 - 2023
Arnaud Blouin -- arnaud.blouin@irisa.fr

Menu

- Introduction
  - Compétences visées par ce cours
  - Compétences non visées par ce cours
  - Architecture et vocabulaire
  - Évolutions des technologies du Web
  - Outils à utiliser / installer
- REST: Communication front-end -- back-end
  - Introduction
  - Les requêtes REST
  - Les requêtes REST avec Spring Boot
  - Données des requêtes REST
  - Réponse HTTP
  - DTO
  - Marshalling _ OpenAPI
  - Service
  - Gestion des exceptions
- ORM: Communication back-end -- base de données
  - Repository
  - JPA
- Test des routes REST
- Sécurité des back-end
- Les langages de données : XML, JSON et YAML
- Point réseau
- 4INFO : front-end

Compétences visées par ce cours

- Savoir fabriquer un back-end (Java)
  - Savoir définir des routes REST
  - Savoir tester des routes REST (et un back de manière générale)
  - Savoir mettre en place une authentification
  - Savoir définir et utiliser du marshalling, avec une API DTO
  - Savoir lier un back-end avec une base de données avec un ORM
  - Savoir concevoir son API REST avec la spécification OpenAPI 3.1
- Savoir comment une application Web s'exécute sur un serveur
- Connaître les différents concepts et technologies du Web actuel

Compétences non visées par ce cours

- Fabriquer et tester un front-end (4 et 5 INFO)
- Fabriquer une base de données (cf l'autre partie de BDWEB)
- Déployer automatique d'applications Web (DevOps) (5 INFO)
- Maîtriser le protocole HTTP (Réseau, 3 INFO)

Architecture et vocabulaire

Page Web vs Application Web

Page Web

- Le terme page Web implique souvent un contenu HTML/CSS/JavaScript basique et statique

Application Web

- Un logiciel, ou en fait un groupement de logiciels, accessible via Internet
- logiciel non installé sur votre machine (ce que l'on appelle un logiciel distribué ou *Cloud-native*)
- Souvent: une séparation client-serveur
  - la partie client = la partie graphique (front-end)
  - la partie serveur = un ensemble de services Web (base de données, calculs, CDN, pub, etc.) avec lesquels le front-end communique
- Souvent: une page Web est en faite une application Web simple

- Questions philosophiques :
  - est-ce qu'une appli mobile est une appli Web ?
  - est-ce qu'une appli bureau qui communique avec des services Web est une appli Web ?

Architecture et vocabulaire

Page Web ou Application Web ?

- https://chat.insa-rennes.fr
- http://people.irisa.fr/Arnaud.Blouin/
- https://www.insa-rennes.fr
- https://www.netflix.com

Regardez les sources des pages Web (ctrl+U sous Firefox) ou inspecter la page (clic-droit -> inspecter) !

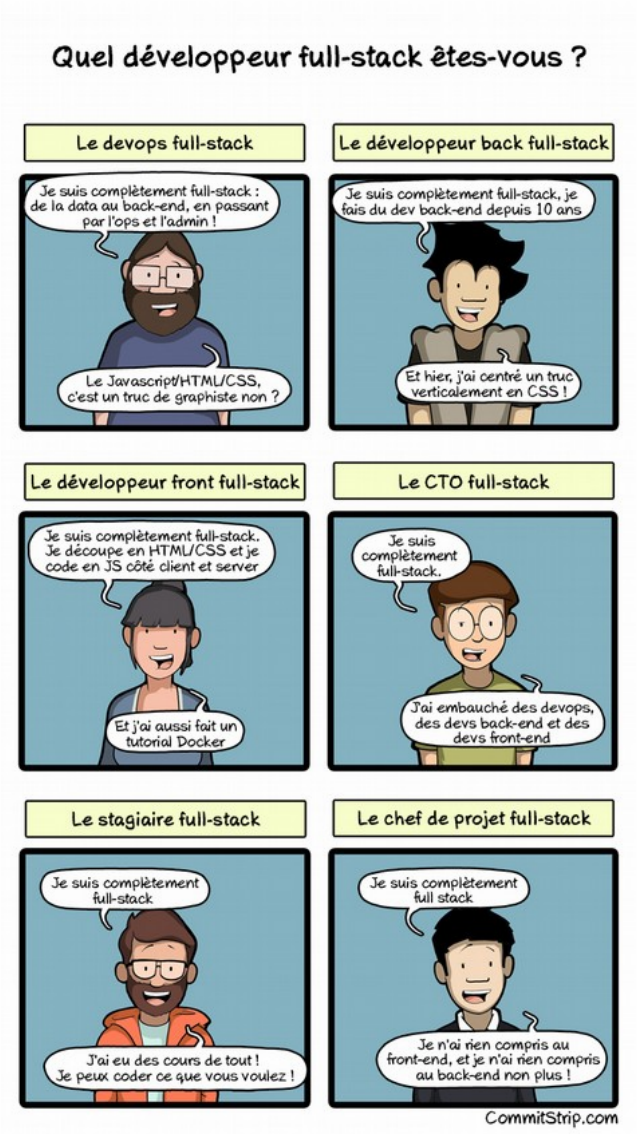
https://fr.wikipedia.org/wiki/Syst%C3%A8me_de_gestion_de_contenu
(à ne pas confondre avec système gestion de versions)

Un peu de Web en images

Front-office: front-end côté entreprise (administration, statistiques, etc.)

Back-office: back-end côté entreprise

Un peu de Web en images



- **stack**: pile des technologies utilisées
- **full-stack**: décrit généralement une personne pouvant développer toutes les parties (e.g., front-end, back-end) d'une application Web

Un peu de Web en images

Outils utilisés (et à installer)

- Les vrais sont sous Linux. En ce qui concerne les autres, vous pouvez toujours vous y mettre.
- Vérifier sa version de (Java 17) : `java -version`
- Vérifier que Maven est installé (Maven 3) : `mvn -v`
- Avoir Swagger editor en local : https://github.com/swagger-api/swagger-editor
  Le plus simple est d'installer docker et de lancer les commandes suivantes (mettre un `sudo` devant chaque commande si demandé) :

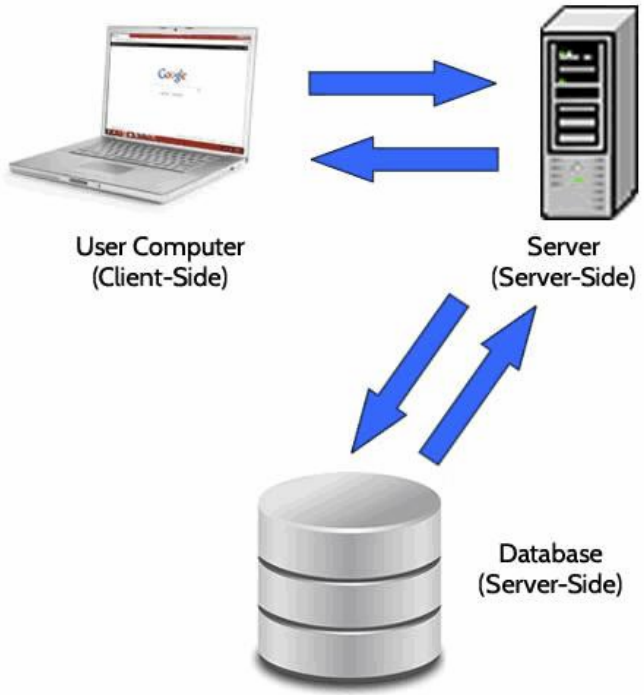
  ```
  docker pull swaggerapi/swagger-editor
  docker run -d -p 1024:8080 swaggerapi/swagger-editor
  ```

  et dans votre navigateur aller sur la page `http://localhost:1024/` À tout moment vous pouvez retrouver votre instance docker Swagger avec `docker ps -a`. La première colonne affichée vous indique l'ID de l'instance. Vous pouvez la stopper ou la redémarrer (à chaque début de TP) avec `docker start <id>` et `docker stop <id>`
- Avoir IntelliJ ou VisualCode

Client-server communication: REST Architecture style

REST

- **REST**: Representational State Transfer (2000)
- **An architecture style**
- A set of principles to leverage, follow Web architectural principles
  - Client – server separation
  - Stateless commands
  - cacheable
  - uniform interface
- Most of the REST libraries use the **HTTP protocol**
- Not limited to Java: REST is a pattern architecture style implemented in various languages
  (we will also practice NestJS and TypeScript)



REST : Pourquoi ?

- **Permet de réaliser des opérations CRUD (Create-Read-Update-Delete)**, avec **authentification**
  - Ne pas oublier que les données sont stockées dans des bases de données côté serveur et non côté front-end. Donc il est nécessaire pour les front-ends d'interroger des back-ends pour obtenir ces données.

- À votre avis, comment les données sont formatées/transmises entre deux services Web ou entre un front et un back ?
  - Est-ce qu'un front-end peut récupérer un objet Java tel quel ?
  - Est-ce que les services Web font partie d'un même programme ? Utilisent les même technologies ? Sont localisés au même endroit ?

(Ces questions et réponses s'appliquent également aux autres techniques de communication entre services Web, exemple WebSockets)

REST : Avantages / inconvénients

Pour :

- Permet de réaliser des opérations CRUD (Create-Read-Update-Delete), avec authentification
- Repose sur le protocole HTTP
- Asynchrone
- Permet de créer une API REST (Application Programmation Interface), ie un dialect explicit entre les front-ends et le back-end
- Géré par la plupart bibliothèques Web (Angular, React, JavaScript, Spring, NestJX, etc.)

Contre :

- Asynchrone (exemple : problématique pour jeu vidéo à très basse latence)
- Peut être verbeux/volumineux (header HTTP, etc.) (exemples : problématique pour jeu vidéo à très basse latence, un appareil basse consommation, une très faible bande passante)
- Que dans un sens : le back-end ne connait pas les front-ends (donc de communication back-end vers front-end)

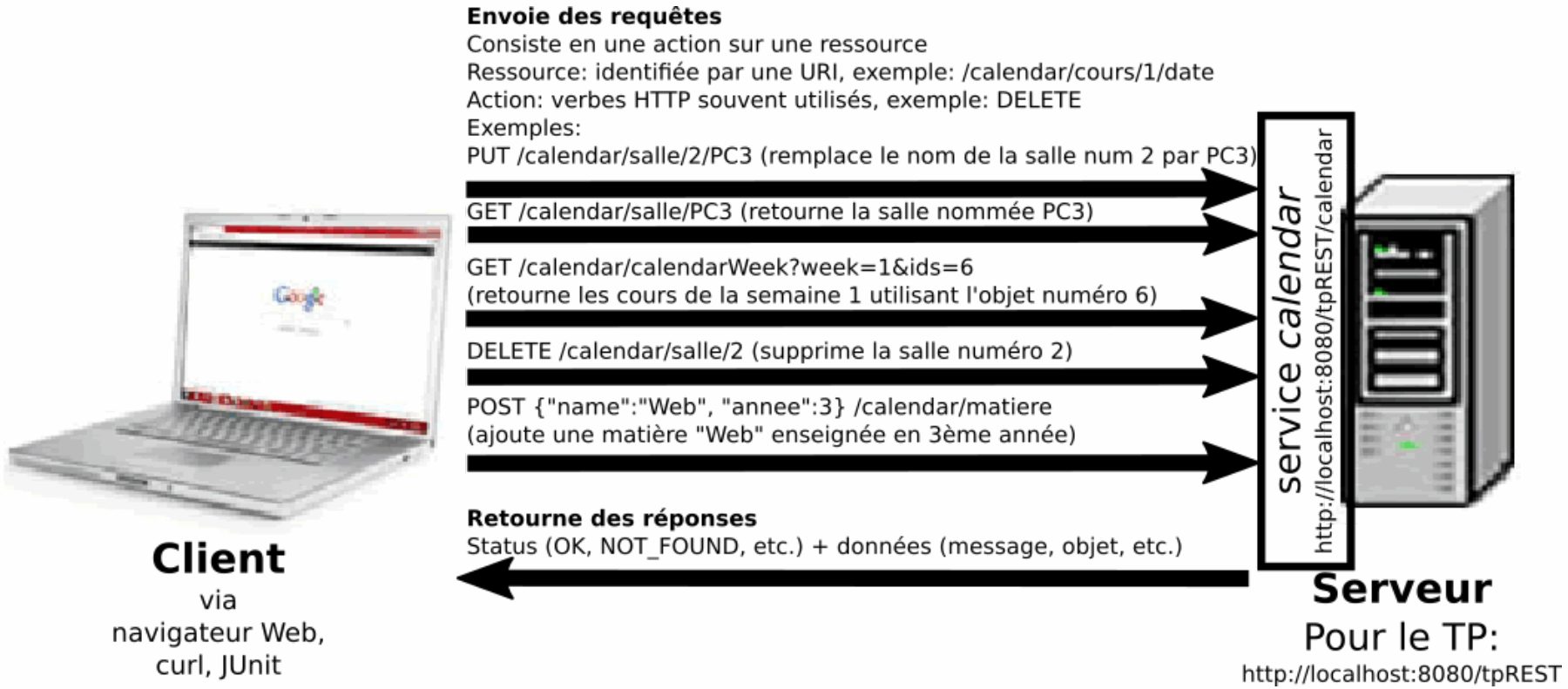
Autres technologies :

- WebSockets: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API

REST

Main concepts

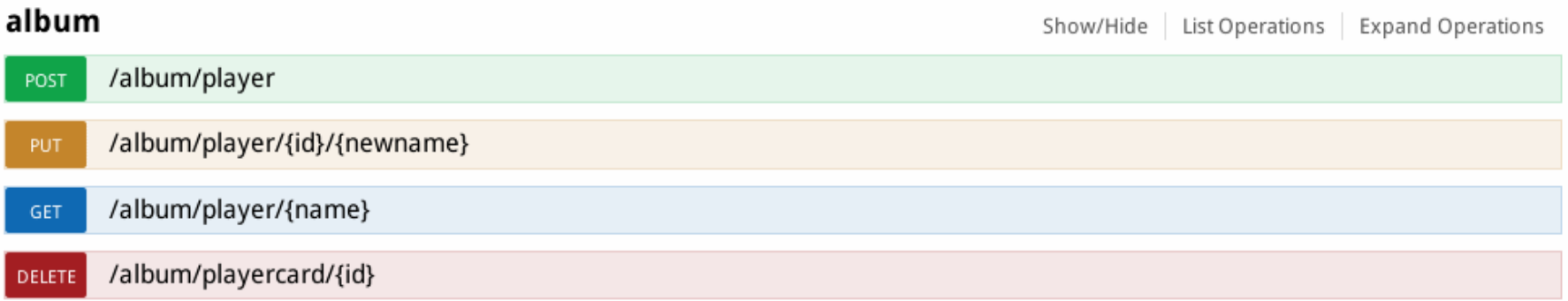
- **Sets of resources** on server side
- **Resource ID** (URI) to identify resources
- **Transfered data**
- **Action** to interact with a resource



REST : vocabulaire

- **Ressource REST / Controleur REST** : objet du back-end (exemple : une classe Java) qui définit un ensemble de routes REST. Un back-end REST peut définir plusieurs resources REST. Une ressource REST est identifiée par un morceau d'URI (exemple : `api/v2`)
- **Route/requête REST** : Une route/requête REST pointe au final vers une méthode du back-end. Contient un verbe, une URI, des données d'entrée, une réponse HTTP, etc. En anglais: **REST route, REST endpoint**.
- **Requête HTTP versus Reqûete REST** : REST utilise HTTP. Un requête REST est un sur-ensemble d'une requête HTTP
- **Réponse HTTP** : cf cours de réseau. Toute requête HTTP retourne un réponse HTTP. Dans Firefox, ouvrez la console de dev (Ctrl+Shift+I) et allez dans l'onglet réseau, rechargez la page, cliquez sur une ligne : vous pouvez voir à droite l'en-tête, la réponse, les cookies, etc, de la requête HTTP.
- **Verbe/Action** : `POST`, `GET`, `DELETE`, `PUT`, `PATCH` (proviennent de HTTP). Décrit le type d'action que va faire la requête
- **URI** : une chaîne de caractère du type : `foo/bar/yolo`. Une URL est une URI. Ainsi, le début de l'URL est l'emplacement du back-end (`https://monadresse.fr`) et vient ensuite l'URI des routes REST (`https://monadresse.fr/api/v2/patient`). Une route REST ne décrit que son URI (ici `patient`) qui vient s'ajouter à l'URI de sa ressource (`api/v2`), qui vient s'ajouter à l'adresse du serveur.

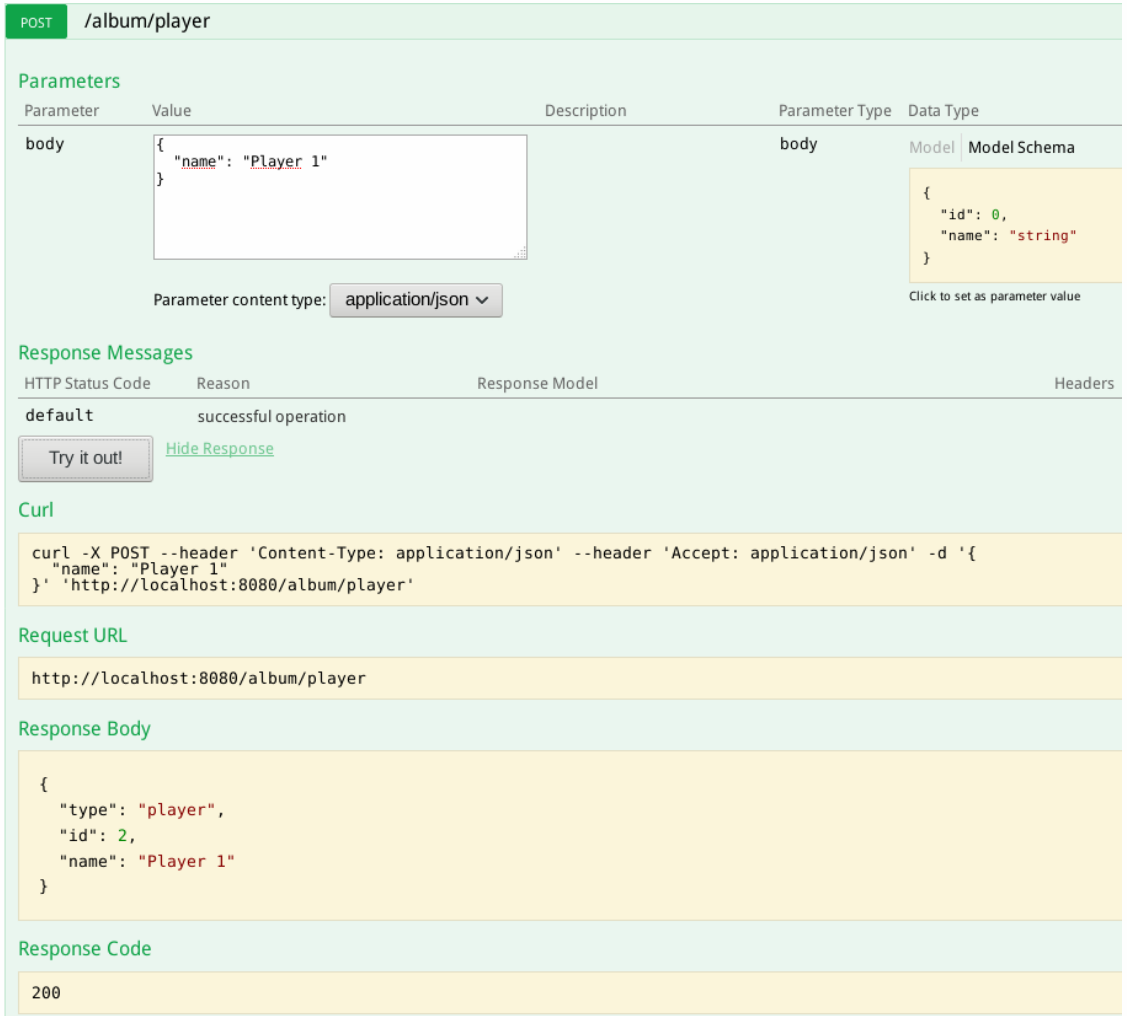
REST : exemple



- l'URI de la ressource REST est `album`
- Elle contient 4 routes REST (attention : les URI affichées inclues l'URI de la ressource)
- **Chaque route REST a un verbe d'action, une URI** (avec des paramètres pour certaines)
  - Verbe + URI = tuple unique. Sinon le service ne sait pas quelle route prendre
  - Exemples :
    - `POST foo/bar` et `GET foo/bar` ne sont pas en conflits
    - `GET foo/bar/{param1}` et `GET foo/bar/{param2}` sont en conflits (car le nom des paramètres ne compte pas)
- On ne le voit pas, mais **on peut aussi définir la réponse** (code HTTP, données, message, etc.), **les cookies**, etc. (cf slide suivant)

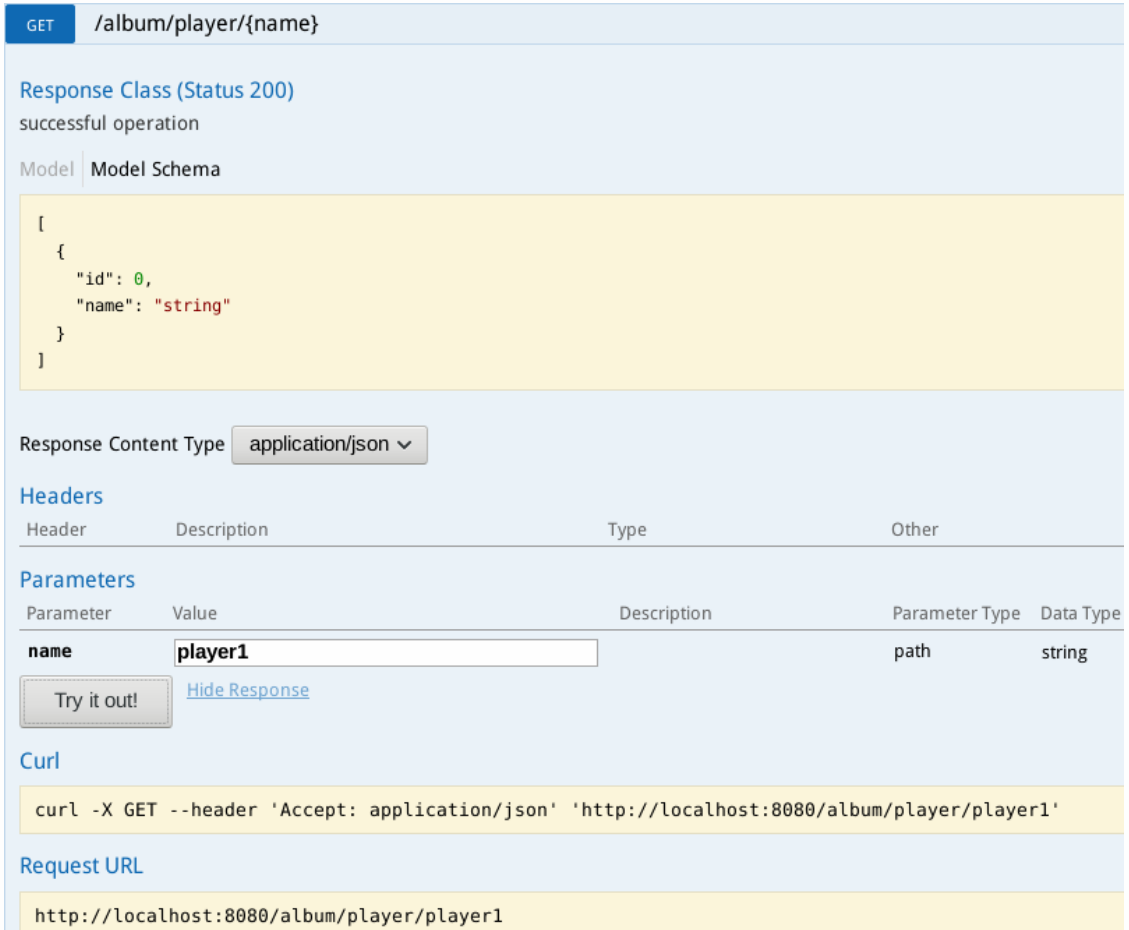
REST : POST

- **Création de nouvelles données**
- Bonnes pratiques du POST :
  - **La donnée à ajouter est placée dans le corps de la requête** (le `body`) (on verra ensuite les différentes manières de transmettre des données via une requête REST)
    - **Pourquoi** : données volumineuses ; pollution visuelle de l'URL; etc.
  - Généralement peu voire pas de paramètres dans l'URI de la requête
- **À noter** : le code de retour et le corps de la réponse
- **Curl** : un outil pour exécuter des requêtes HTTP



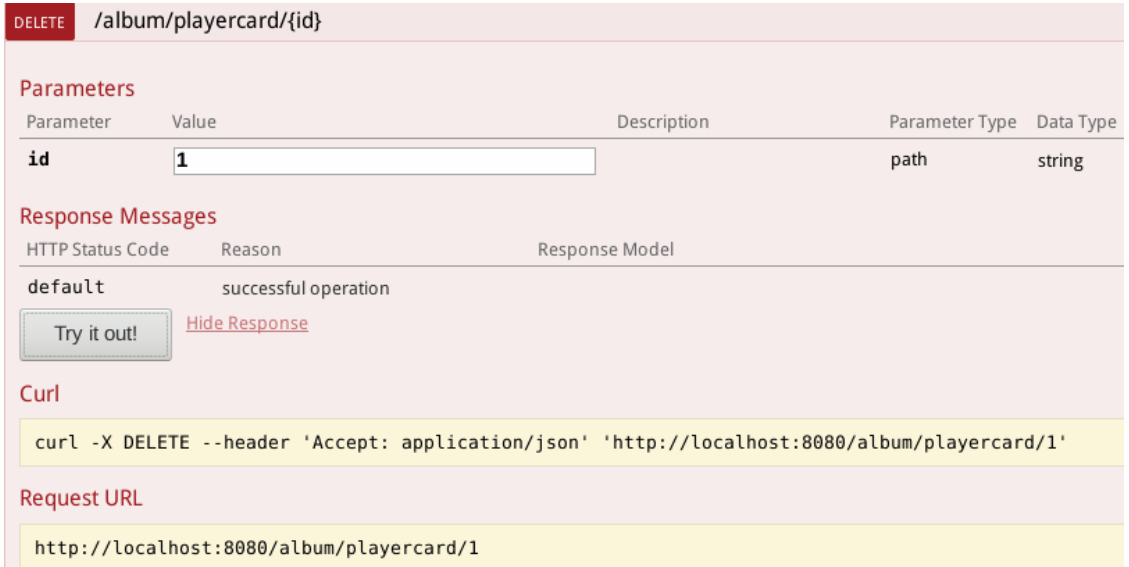
REST : GET

- **Récupération de données existantes**
- Bonnes pratiques du GET :
  - **Lecture seule**



REST : DELETE

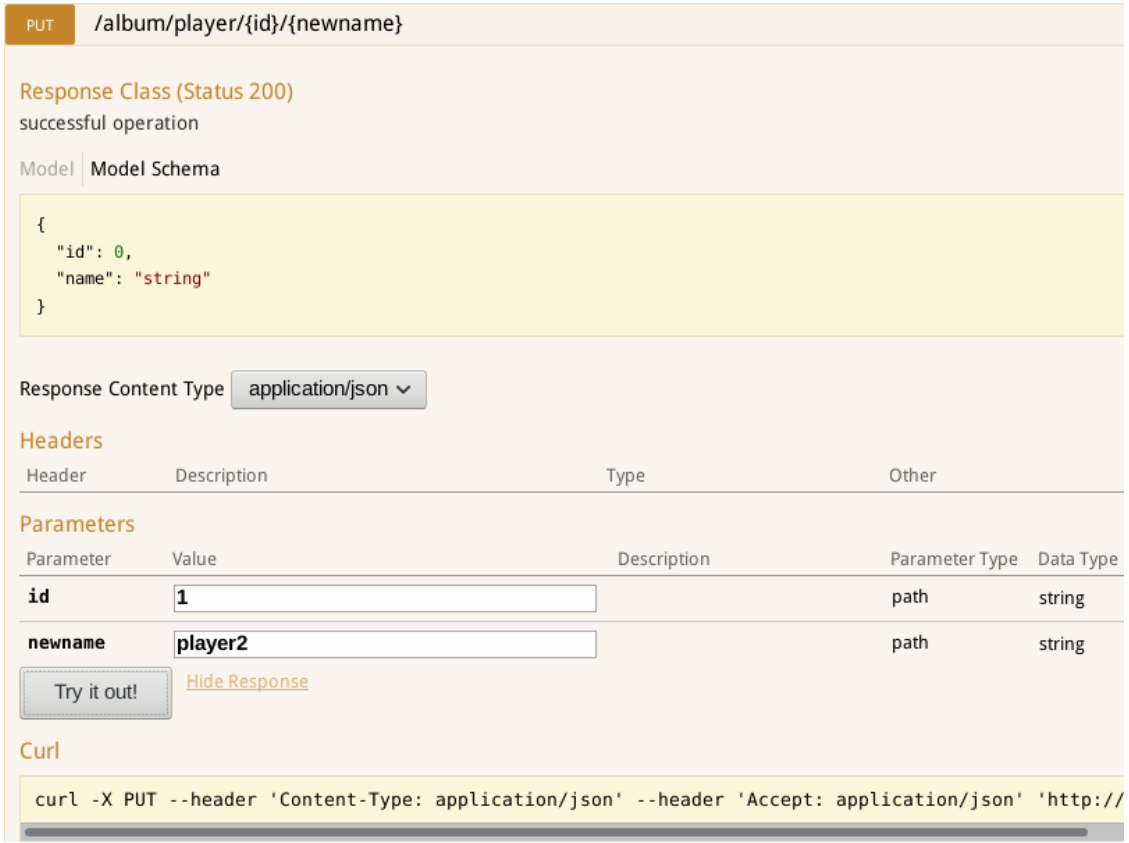
- **Suppression de données existantes**



REST : PUT et PATCH

- En théorie :
  - `PUT` : **Remplacement d'une donnée existante par une autre**
  - `PATCH` : **Modification d'une donnée existante**

- En pratique :
  - `PATCH` a été ajouté après les autres verbes, et est donc peu utilisé
  - Très souvent on utilise `PUT` pour aussi faire du `PATCH`



Outils pour faire développer un back-end REST

- Bibliothèques Java :
  - *Spring Boot* (https://spring.io/projects/spring-boot)
  - *Jersey* (https://github.com/eclipse-ee4j/jersey/wiki)
  - *Apache CXF*, *RESTeasy*, etc.

Qu'est ce que **JAX-RS**, **Jakarta** ?
L'écosystème Java a conçu une API pour définir des back-ends REST : **JAX-RS** (*Java API for RESTful Web Services*). JAX-RS est désormais l'acronyme de *Jakarta RESTful Web Services* (on parle donc des fois de *Jakarta*). JAX-RS est une API : un ensemble d'interfaces et d'annotations Java pour faire du REST. L'avantage de JAX-RS est que l'on peut coder son back-end juste avec cette API, pour ensuite configurer l'implémentation qui va la faire fonctionner. On peut donc changer (presque) facilement d'implémentation.
Jersey, *Apache CXF*, *RESTeasy* sont des implémentations de JAX-RS. *Spring Boot 2* n'utilise pas JAX-RS. *Spring Boot 3* utilise JAX-RS (nous allons l'utiliser).

- Bibliothèques TypeScript :
  - *NestJS* (https://nestjs.com)

- Une multitude de bibliothèques dans chaque langage

- Pour décrire une API REST : *OpenAPI* (https://www.openapis.org) avec l'outil *https://editor.swagger.io*
- Pour tester une API REST : *Postman* (https://www.postman.com)

Spring Boot

- https://start.spring.io pour créer un projet
- Démo : https://github.com/arnobl/WebEngineering-INSA/tree/master/rest/springboot3
- Pour nous, en Java 17, avec Maven, Spring Boot 3
  - Maven : gestion des dépendances, compilations, exécution des tests, etc.
  - J'ai ajouté des plugins : `spring-boot-starter-web` pour faire du REST, `jackson-dataformat-xml` pour marshaller en XML, `lombok` pour produire des constructeurs, des getters, setters à l'aide d'annotations
- Le `Main` est simple et est directement executable :

```
@SpringBootApplication
public class WebApplication {
    public static void main(String[] args) {
        SpringApplication.run(WebApplication.class, args);
    }
}
```

- Lorsque vous lancer le `Main`, vous démarrez le serveur back-end

Spring Boot : créer une ressource REST

- Dans le package principal, créer un nouveau package `restcontrollers` (en Spring une ressource REST s'appelle un contrôleur REST)
- Y ajouter une nouvelle classe `HelloControllerV1`

```
@RestController
@RequestMapping("api/public/v1/hello")
public class HelloControllerV1 {
}
```

- Annotation `@RestController` : marque la classe comme étant une ressource REST
- Annotation `@RequestMapping` : définit l'URI de la ressource REST (ici `api/public/v1/hello`)
- Lors du démarrage du programme, Spring trouve automatiquement les ressources REST grâce à `@RestController`

Spring Boot : définir une route REST GET

```java
@RestController
@RequestMapping("api/public/v1/hello")
public class HelloControllerV1 {

    @GetMapping(path = "world", produces = MediaType.TEXT_PLAIN_VALUE)
    public String helloWorld() {
        return "Hello world!";
    }
}
```

- Une route REST est une méthode dans la classe de sa ressource REST
- Le nom de la méthode a peu d'importance
- L'annotation `@GetMapping` définit que la route est un `GET`
  - Son paramètre `path` définit son URI
  - Le paramètre `produces` définit le type des données retournées, ici du texte brut
- Avec votre navigateur, allez sur http://localhost:8080/api/public/v1/hello/world
  Allez sur une URL dans un navigateur est une requête HTTP GET

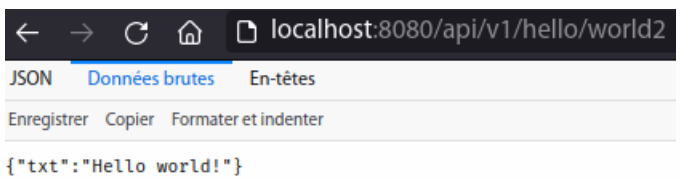
Spring Boot : définir une route REST GET

Retour en JSON ou en XML

```java
@RestController
@RequestMapping("api/public/v2/hello")
public class HelloControllerV2 {
    @GetMapping(path = "world", produces = MediaType.APPLICATION_JSON_VALUE)
    public Message helloWorld() {
        return new Message("Hello world!");
    }
}

record Message(String txt) {}
```

- En Java 17, un `record` est une classe de données en lecture seule
- Dans notre navigateur, on obtient alors :



- Pour produire de l'XML : `MediaType.APPLICATION_XML_VALUE`

Spring Boot : définir une route REST POST

```java
@RestController
@RequestMapping("api/public/v1/hello")
public class HelloControllerV1 {
    private final Set<String> txts;

    public HelloController() {
        super();
        txts = new HashSet<>();
        txts.add("foo");
        txts.add("bar");
    }

    @PostMapping(path = "txt", consumes = MediaType.TEXT_PLAIN_VALUE)
    public void newTxt(@RequestBody final String txt) {
        txts.add(txt);
    }
}
```

- En ligne de commande : `curl -X POST "http://localhost:8080/api/public/v1/hello/txt" -H "Content-Type: text/plain" -d "foo"`
  Ca n'affiche rien (bonne nouvelle, pas de nouvelle) car `void`. On verra plus tard comment personnaliser la réponse

Spring Boot : définir une route REST POST

- Avec du JSON ou XML en entrée

```
@RestController
@RequestMapping("api/public/v2/hello")
public class HelloControllerV2 {
//...

// ou MediaType.APPLICATION_XML_VALUE
    @PostMapping(path = "txt", consumes = MediaType.APPLICATION_JSON_VALUE)
    public void newTxt(@RequestBody final Message txt) {
        txts.add(txt.text());
    }
}
```

- En ligne de commande :

```
curl -X POST "http://localhost:8080/api/public/v2/hello/txt" -H "accept: application/json" -H "Content-Type: application/json" -d '{ "text": "foo"}'
```

Spring Boot : définir une route REST DELETE

```java
@RestController
@RequestMapping("api/public/v1/hello")
public class HelloControllerV1 {
//...

    @DeleteMapping(path = "txt/{txtToRemove}")
    public void removeTxt(@PathVariable("txtToRemove") final String txt) {
        txts.remove(txt);
    }
}
```

- En ligne de commande :

```
curl -X DELETE "http://localhost:8080/api/public/v1/hello/txt/foo"
```

- Pas de conflit avec la route POST car même si les URI sont identiques, les verbes sont différents
- Avec du JSON ou XML en entrée, suivre la même méthode que le slide précédent

Spring Boot : définir une route REST PATCH

Version par terrible

```java
@RestController
@RequestMapping("api/public/v1/hello")
public class HelloControllerV1 {
    private User user = new User("Foo", "here", "1");

    @PatchMapping(path = "user",
    consumes = MediaType.APPLICATION_JSON_VALUE)
    public void patchUser(@RequestBody final User patchedUser) {
        user.patch(patchedUser);
    }
}

class User {
    private String name;
    private String address;
    private final String id;

  //...

    public void patch(final User user) {
        if(user.address != null) {
            address = user.address;
        }
        if(user.name != null) {
            name = user.name;
        }
    }
}
```

- Avec `PATCH` la route reçoit un objet partiel (les attributs non changés ne sont pas définis dans le JSON et donc null dans l'objet `patchedUser` reçu)
- L'avantage de `PATCH` est que l'on peut modifier plusieurs attributs en une seule requête : réduction du nombre de requêtes à définir et à utiliser
- La méthode `patch` dans la classe `User` copie les attributs non nuls et mutables (évidemment ca ne fonctionne pas si l'on veut mettre à nul un attribut)
- **Cette version n'est pas idéale. Pourquoi ?**

```
curl -X PATCH "http://localhost:8080/api/public/v1/hello/user" -H "accept: application/json" -H "Content-Type: application/json" -d '{ "name": "aa"}'
```

Spring Boot : définir une route REST PATCH

Version améliorée

```
@RestController
@RequestMapping("api/public/v1/hello")
public class HelloControllerV1 {
    @Autowired private UserService userService;

    @PatchMapping(path = "user/{id}",
    consumes = MediaType.APPLICATION_JSON_VALUE)
    public void patchUser(@RequestBody Map<String, Object> partialUser,
                          @PathVariable("id") final long id) {
        userService.patchUser(partialUser, id);
    }
}

@Service public class UserService {
  @Autowired private ObjectMapper objectMapper;

  public Optional<User> patchUser(
                      Map<String, Object> partialUser, long id) {
        Optional<User> userOpt = findUser(id);
        if(userOpt.isEmpty()) {
            return Optional.empty();
        }
        User user = userOpt.get();
        try {
            objectMapper.updateValue(user, partialUser);
        } catch(JsonMappingException ex) {
            return Optional.empty();
        }
        return userOpt;
}}
```

- La route patch ne prend plus un objet `User` mais une table de hashage contenant les attributs du JSON reçu.
- Pour simplifier, la route a maintenant un attribut `id` pour identifier le `User` concerné
- `objectMapper` est un objet qui s'occupe du marshalling dans le back-end.
- `objectMapper` possède une méthode `updateValue` qui peut prendre une table de hashage de valeurs et modifie l'objet
- À noter qu'il existe aussi une autre méthode pour faire un PATCH avec JSON-PATCH https://jsonpatch.com

Spring Boot : définir une route REST PUT (en théorie)

```java
@RestController
@RequestMapping("api/public/v1/hello")
public class HelloControllerV1 {
    private User user = new User("Foo", "here", "1");

    @PutMapping(path = "user",
      consumes = MediaType.APPLICATION_JSON_VALUE)
    public void replaceUser(@RequestBody final User patchedUser) {
        user = patchedUser;
    }
}
```

- En théorie, avec PUT la route reçoit un objet complet et remplace l'objet ciblé (après vérification), contrairement à PATCH

Commande Curl:

```
curl -X PUT "http://localhost:8080/api/public/v1/hello/user" -H "accept: application/json" -H "Content-Type: application/json" -d '{ "name": "aa", "id":"2",

"address":"there"}'
```

Spring Boot : définir une route REST PUT (en pratique)

```java
@RestController
@RequestMapping("api/public/v1/hello")
public class HelloControllerV1 {
    private User user = new User("Foo", "here", "1");

    @PutMapping(path = "rename/{newname}")
    public void renameuser(@PathVariable("newname") String newname) {
        user.setName(newname);
    }
}
```

- Comme dit précédemment, souvent `PUT` est utilisé pour modifier un objet
- Ici, on renomme le `User`. Attention : si tous les paramètres d'un `User` sont modifiables, pensez à la technique vue avec `PATCH` (et ne pas faire une route REST par attribut à modifier)
- Le nouveau nom est inclus dans l'URI de la route REST (entre accolade). C'est un paramètre de l'URI.

Commande Curl:

```
curl -X PUT "http://localhost:8080/api/public/v1/hello/rename/foo"
```

REST : les données des requêtes

Il existe trois manières de passer des données dans une requête

- **Dans le corps de la requête HTTP (le body)**

```java
@PatchMapping(path = "user", consumes = MediaType.APPLICATION_JSON_VALUE)
public void patchUser(@RequestBody User patchedUser) {
  user.patch(patchedUser);
}
```

```
curl -X PATCH "http://localhost:8080/api/public/v1/hello/user" -H "Content-Type: application/json" -d '{ "name": "aa"}'
```

- **Dans l'URI de la route REST**

```java
@PutMapping(path = "rename/{newname}")
public void renameuser(@PathVariable("newname") String newname) {
  user.setName(newname);
}
```

- **Dans les paramètres de l'URI**

`http://localhost:8080/api/public/v1/hello/user?newname=foo` (et non `api/public/v1/hello/user/foo`)

```java
@PutMapping(path = "rename")
public void renameuser(@RequestParam("newname") String newname) {
  user.setName(newname);
}
```

REST : les données des requêtes

Dans le corps de la requête HTTP (le body)

Avantages

- Données complexes
- Cacher un peu les données
- JSON, XML, etc.

Inconvénients

- Plus compliqué à utiliser

Dans l'URI de la route REST

Avantages

- Simple à utiliser

Inconvénients

- Ne fonctionne pas avec des données complexes
- Pollution de l'URL
- Affichage en clair de données

REST : la réponse HTTP d'une requête REST

```
@RestController
@RequestMapping("api/public/v1/hello")
public class HelloControllerV1 {
//...

    @DeleteMapping(path = "txt", consumes = MediaType.TEXT_PLAIN_VALUE)
    public void removeTxt(@RequestBody final String txt) {
        txts.remove(txt);
    }
}
```

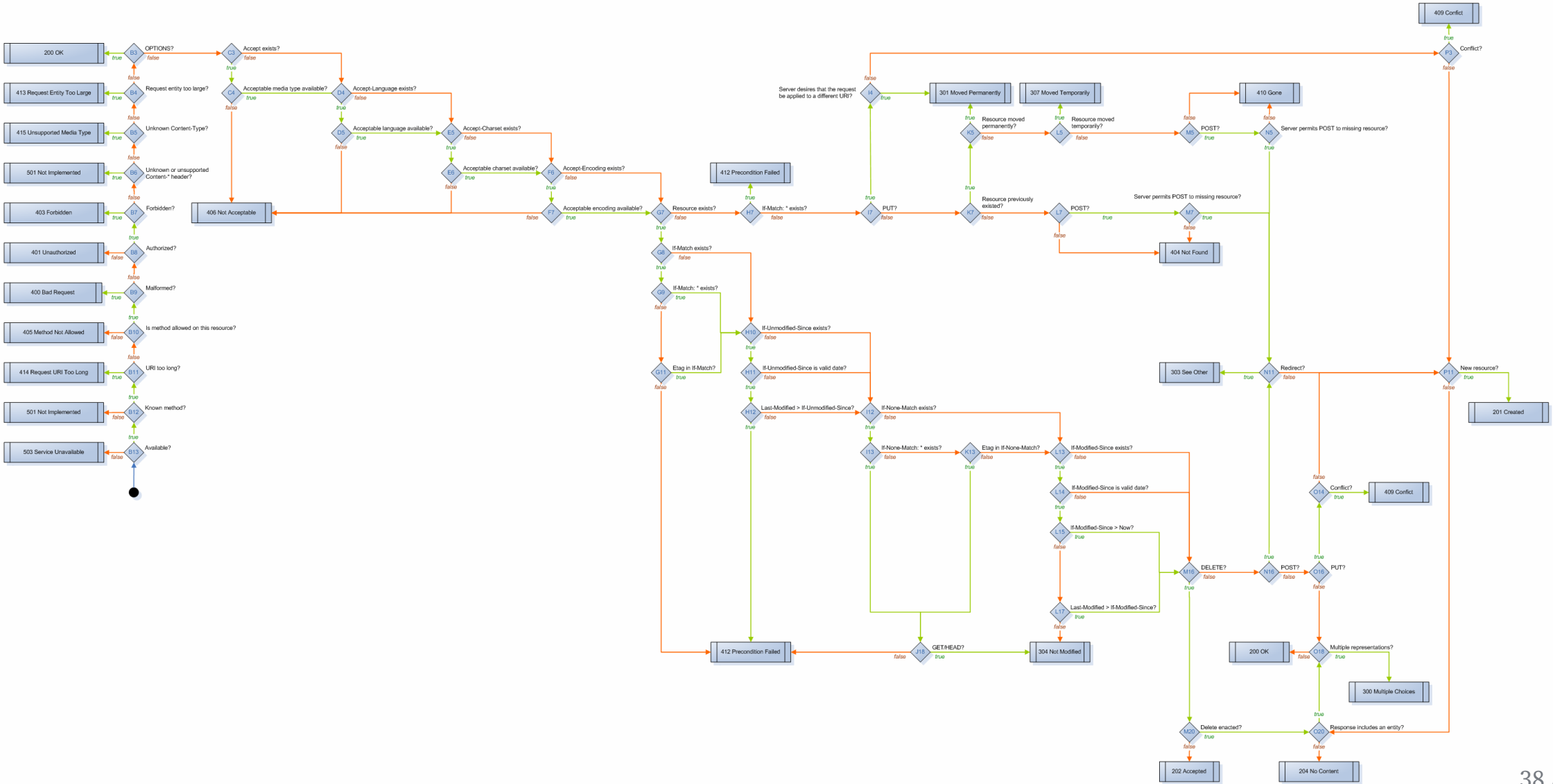
- Question : est-ce que le front-end reçoit `void` ?

REST : la réponse HTTP d'une requête REST

```
@RestController
@RequestMapping("api/public/v1/hello")
public class HelloControllerV1 {
//...

    @DeleteMapping(path = "txt", consumes = MediaType.TEXT_PLAIN_VALUE)
    public void removeTxt(@RequestBody final String txt) {
        txts.remove(txt);
    }
}
```

- En fait le back-end retourne toujours une réponse HTTP en réponse à une requête REST
- Par simplicité, on peut mettre `void`, mais le programme va en fait retourner une réponse HTTP sans corps avec un code de retour HTTP
- Réponse HTTP : code de retour (cf slide suivant), éventuellement des données, un message, etc.


200 OK
B3 OPTIONS?
413 Request Entity Too Large
B4 Request entity too large?
415 Unsupported Media Type
B5 Unknown Content-Type?
501 Not Implemented
B6 Unknown or unsupported Content-* header?
403 Forbidden
B7 Forbidden?
401 Unauthorized
B8 Authorized?
400 Bad Request
B9 Malformed?
405 Method Not Allowed
B10 Is method allowed on this resource?
414 Request URI Too Long
B11 URI too long?
501 Not Implemented
B12 Known method?
503 Service Unavailable
B13 Available?
C3 Accept exists?
C4 Acceptable media type available?
406 Not Acceptable
D4 Accept-Language exists?
D5 Acceptable language available?
E3 Accept-Charset exists?
E6 Acceptable charset available?
F6 Accept-Encoding exists?
F7 Acceptable encoding available?
G7 Resource exists?
412 Precondition Failed
H7 If-Match: * exists?
G8 If-Match exists?
G9 If-Match: * exists?
G11 Etag in If-Match?
H10 If-Unmodified-Since exists?
H11 If-Unmodified-Since is valid date?
H12 Last-Modified > If-Unmodified-Since?
I12 If-None-Match exists?
I13 If-None-Match: * exists?
K13 Etag in If-None-Match?
L13 If-Modified-Since exists?
L14 If-Modified-Since is valid date?
L15 If-Modified-Since > Now?
L17 Last-Modified > If-Modified-Since?
J18 GET/HEAD?
304 Not Modified
I4 Server desires that the request be applied to a different URI?
301 Moved Permanently
307 Moved Temporarily
K5 Resource moved permanently?
L5 Resource moved temporarily?
410 Gone
M5 POST?
N5 Server permits POST to missing resource?
I7 PUT?
K7 Resource previously existed?
L7 POST?
M7 Server permits POST to missing resource?
404 Not Found
409 Conflict
P3 Conflict?
303 See Other
N11 Redirect?
P11 New resource?
201 Created
O14 Conflict?
409 Conflict
M16 DELETE?
N16 POST?
O16 PUT?
200 OK
O18 Multiple representations?
300 Multiple Choices
M20 Delete enacted?
O20 Response includes an entity?
202 Accepted
204 No Content

REST : la réponse HTTP d'une requête REST

Code de retour

Le code de retour est très important. Il définit le résultat de l'opération, exemples :

- `404 - NOT FOUND`
- `200 - OK`
- `500 - Internal Serveur Error` (erreur dans le back-end)
- `204 - No Content` (souvent lorsque la route retourne `void`)
- `400 - Bad Request`
- `405 - Method Not Allowed` (mauvais verbe sur la route)

Exemple

```
curl -X POST "http://localhost:8080/api/public/v3/hello/world" -i
HTTP/1.1 405
Allow: GET
```

Cette route n'accepte pas de `POST` mais des `GET`. Donc `405` (*Method Not Allowed*)

REST : la réponse HTTP d'une requête REST

Choisir le code de retour

```java
@PostMapping(path = "txt", consumes = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<Void> newTxt(@RequestBody final Message txt) {
    if(txts.add(txt.text())) {
        // return new ResponseEntity<>(HttpStatus.OK);
        return ResponseEntity.ok().build();
    }
  // return new ResponseEntity<>(HttpStatus.BAD_REQUEST);
  return ResponseEntity.badRequest().build();
}
```

Exécution deux fois de:

```
curl -X POST "http://localhost:8080/api/public/v3/hello/txt" -H "Content-Type: application/json" -d '{ "text": "foos"}' -i
```

Résultat 1 :

```
HTTP/1.1 200
```

Résultat 2 :

```
HTTP/1.1 400
```

À noter que l'on peut créer des réponses de deux manières différentes (cf les exemples ci-dessus)

REST : réponse HTTP avec données

Exemples

```java
@GetMapping(path = "you", produces = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<Message> helloYou() {
  // return new ResponseEntity<>(new Message("Hello you!"), HttpStatus.OK);
  return ResponseEntity.ok(new Message("Hello you!"));
}
```

```
curl -X GET "http://localhost:8080/api/public/v3/hello/you" -i
HTTP/1.1 200
Content-Type: application/json
{"text":"Hello you!"}
```

```java
@GetMapping(path = "txts", produces = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<Set<String>> getAllTxts() {
  //return new ResponseEntity<>(txts, HttpStatus.OK);
  ResponseEntity.ok(txts);
}
```

```
curl -X GET "http://localhost:8080/api/public/v3/hello/txts" -i
HTTP/1.1 200
Content-Type: application/json
["bar","foo"]
```

À noter que l'on peut créer des réponses de deux manières différentes (cf les exemples ci-dessus)

REST : DTO (Data Transfer Object)

Exemple problématique

```java
public class User {
    private String name;
    private String address;
    private String id;
    private String pwd;
  //...
}
```

```java
@GetMapping(path = "user", produces = MediaType.APPLICATION_JSON_VALUE)
public User getUser() {
  return user;
}
```

```
curl -X GET "http://localhost:8080/api/public/v1/hello/user" -i
```

```
HTTP/1.1 200
Content-Type: application/json
{"name":"Foo","address":"here","id":"1","pwd":"p1"}
```

Oups... J'ai renvoyé tout l'objet `User` sans faire attention qu'il avait un attribut confidentiel `pwd`

REST : DTO (Data Transfer Object)

- Depuis mon back-end, je voudrais donc répondre à une requête avec des données
- **Mais:**
  - La structure de mon objet n'est pas optimisée pour le transfert
  - La structure de mon objet contient des données sensibles que je veux exclure
- **Solution:**
  - Définir une classe dédiée uniquement à représenter les attributs que l'on veut transferer d'un objet
  - Un tel objet est appelée un DTO (data transfert object)

```java
// Lombok annotations: generates getters and setters
@Getter
@Setter
@NoArgsConstructor
public class UserDTO {
    private String name;
    private String address;
    private String id;

    public UserDTO(final User user) {
        this.name = user.getName();
        this.address = user.getAddress();
        this.id = user.getId();
    }
}
```

```java
@GetMapping(path = "user", produces = MediaType.APPLICATION_JSON_VALUE)
public UserDTO getUser() {
  return new UserDTO(user);
}
```

Résultat :

```
curl -X GET "http://localhost:8080/api/public/v3/hello/user" -i

HTTP/1.1 200
Content-Type: application/json
{"name":"Foo","address":"here","id":"1"}
```

REST : DTO (Data Transfer Object)

- Les DTO s'utilise aussi dans le body d'une requête

```
@PatchMapping(path = "user", consumes = MediaType.APPLICATION_JSON_VALUE)
public void patchUser(@RequestBody final UserDTO patchedUser) {
  // Assuming that only a single user exists
  patchedUser.patch(user);
}
```

```
@Getter
@Setter
public class UserDTO {
    private String name;
    private String address;
    private String id;

    public UserDTO(final User user) {
        super();
        this.name = user.getName();
        this.address = user.getAddress();
        this.id = user.getId();
    }

    public void patch(final User user) {
        if(address != null) {
            user.setAddress(address);
        }
        if(name != null) {
            user.setName(name);
        }
    }
}
```

Dans cet exemple, les conversions `User` <-> `UserDTO` se font dans le DTO. On verra l'an prochain que l'on peut créer des fabriques. Mais à éviter dans la classe `User` : on ne pollue généralement pas les classes de données avec du DTO.

REST : Marshalling

- Le **marshalling** est l'opération qui transforme un objet vers un format plus adapté à la transmission
- Le **démarshalling** est l'opération inverse

- **Exemple**
  - Marshalling : Objet Java -> objet JSON -> texte (body d'une requête)
  - Démarshalling : body d'une requête -> objet JSON (côté front-end) -> objet TypeScript (optionnel car TypeScript gère nativement le JSON)

- **Spring gère nativement le marshalling JSON** (Spring utilise la librairie `Jackson`). Pour XML, il faut ajouter `jackson-dataformat-xml` dans les dépendances (cf `pom.xml`)

- Oui mais : **des fois on veut marshaller différemment de Spring** ou **Spring n'arrive pas à (dé-)marshaller** (ignorer des attributs, etc.)
  - Soit vous utilisez des **DTO** (ce que nous ferons)
  - Soit il est possible de configurer le marshalling automatique des classes à l'aide d'annotations

REST : Marshalling

Exemple problématique

```java
@RestController
@RequestMapping("api/public/v1/animal")
public class AnimalController {
    private List<Animal> animals = new ArrayList<>();

    @PostMapping(path = "", consumes = MediaType.APPLICATION_JSON_VALUE)
    public void newAnimal(@RequestBody Animal animal) {
        animals.add(animal);
    }
}
```

```java
public interface Animal {
    int getAge();

    void setAge(final int age);

    String getName();
}
```

```
curl -X POST "http://localhost:8080/api/public/v1/animal" -H  "Content-Type: application/json" -d '{ "age": "10", "name": "yolo"}' -i
```

Résultat : `HTTP/1.1 500`
Cannot construct instance of `fr.insarennes.demo.model.Animal` (no Creators, like default constructor, exist): abstract types either need to be mapped to concrete types, have custom deserializer, or contain additional type information

Normal, car plusieurs problèmes :

- `Animal` est abstrait (une interface). Le marshaller ne sait pas quelle classe instancier
- La structure JSON reçue n'indique pas le type concret de l'objet à créer

REST : Marshalling

Solution au problème

```java
@JsonSubTypes({
    @JsonSubTypes.Type(value = Cat.class, name = "cat"),
    @JsonSubTypes.Type(value = Dog.class, name = "dog")
})
@JsonTypeInfo(use = JsonTypeInfo.Id.NAME, include = JsonTypeInfo.As.PROPERTY, property = "type")
public interface Animal {
    int getAge();

    void setAge(final int age);

    String getName();
}
```

- `JsonTypeInfo` ajoute un attribut `type` dans le JSON produit qui contiendra le nom de la classe marshallé
- `JsonSubTypes` indique lors du démarshalling quelles sont les classes concrètes
- Ces annotations sont `Json...` mais fonctionnent aussi avec XML
- **Attention** : les annotations dépendent de la bibliothèque de marshalling (ici *Jackson*)
- Ce problème peut également s'appliquer aux DTO (si héritage)

REST : Marshalling

Solution au problème

```java
@RestController
@RequestMapping("api/public/v1/animal")
public class AnimalController {
    private List<Animal> animals;

    public AnimalController() {
        animals = new ArrayList<>();
        animals.add(new Cat(1, "foo"));
        animals.add(new Dog(2, "bar"));
    }

    @PostMapping(path = "",
        consumes = {MediaType.APPLICATION_JSON_VALUE,
                    MediaType.APPLICATION_XML_VALUE})
    public void newAnimal(@RequestBody final Animal animal) {
        animals.add(animal);
    }

    @GetMapping(path="all", produces=MediaType.APPLICATION_JSON_VALUE)
    public List<Animal> getAll() {
        return this.animals;
    }
}
```

```
curl -X POST "http://localhost:8080/api/public/v1/animal" -H  "Content-Type: application/json"
     -d '{ "age": "10", "name": "yolo", "type":"cat"}' -i

HTTP/1.1 200
```

```
curl -X GET "http://localhost:8080/api/public/v1/animal/all"  -i

HTTP/1.1 200
Content-Type: application/json
[{"type":"cat","age":1,"name":"foo"},{"type":"dog","age":2,"name":"bar"},
 {"type":"cat","age":10,"name":"yolo"}]
```

En XML :

```
curl -X POST "http://localhost:8080/api/public/v1/animal" "Content-Type: application/xml"
     -d '<animal type="dog"><age>3</age><name>barr</name></animal>' -i
```

Un nouvel attribut `type` est maintenant présent dans le JSON

REST : Marshalling

Ignorer un attribut

Avec l'annotation `JsonIgnore` à poser sur l'attribut concerné

```
@Getter
@Setter
public class Cat extends AnimalBase {
    @JsonIgnore
    private int notToMarshall;

    public Cat(final int age, final String name) {
        super(age, name);
        notToMarshall = 1;
    }
}
```

- L'annotation est `Json...` mais fonctionne aussi avec XML
- **Attention** : un attribut est marshallé uniquement s'il possède un getter et un setter. Il existe différentes stratégies de marshalling. Celle de base en Spring utilise les getter/setter.

REST : Marshalling

Lecture intéressante sur le marshalling avec Jackson et Spring

- https://www.baeldung.com/category/json/jackson/

En particulier

- https://www.baeldung.com/json-reduce-data-size
- https://www.baeldung.com/jackson-optional
- https://www.baeldung.com/jackson-ignore-null-fields
- https://www.baeldung.com/jackson-collection-array

Concevoir son API REST avec OpenAPI

- **Modéliser = réfléchir avant d'agir**. Pas seulement UML. **OpenAPI permet de modéliser (designer) une API REST** avant de l'implémenter
- Permet aussi de **générer** : un squelette de code pour le back-end ; des tests
- https://editor.swagger.io
- En YAML ou JSON (nous ferons en YAML)

```yaml
openapi: 3.0.3
info:
  title: exemple du cours
  description: Cette API REST correspond à celle du cours de Web en 3INFO.
  version: 1.0.0
servers:
  - url: https://api.example.com/api/v1
# Notre API sera divisée en trois blocs : album, player, playercard
tags: # Some annotations used to document the route descriptions (optional)
  - name: album
    description: Les albums
  - name: player
    description: Les joueurs
```

Concevoir son API REST avec OpenAPI

Définir les routes

```yaml
openapi: 3.0.3
...

paths: # Vos routes
  /votreURI: # Toutes les routes avec cette URI
    votreVerbe: # La route avec l'URI et ce verbe
      tags:
        - leTagCorrespondant
      summary: Résumé
      description: Description
      operationId: nomDeLaMéthode
      responses: # la réponse HTTP
        'leCode':
          description: Texte descriptif
          content: # Le contenu de la réponse
            application/json: # si JSON
              schema: # Le format de la donnée
                $ref: '#/components/schemas/leNomDeVotreDTO'
        'autreCode':
          ...
```

```yaml
openapi: 3.0.3
...

paths:
  /player/{playerID}:
    get:
      tags:
        - player
      summary: Returns an existing player
      description: Returns an existing player given an ID
      operationId: getPlayer
      parameters:
        - name: playerID
          in: path
          description: The player ID to get
          required: true
          schema:
            type: integer
            format: int64
      responses:
        '200':
          description: Successful operation
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/PlayerDTO'
        '400':
          description: Invalid ID supplied
```

Concevoir son API REST avec OpenAPI

```yaml
...
paths:
  /player/{playerID}:
    get: # Une route GET
      ...
    delete: # mais aussi une route DELETE
      tags:
        - player
      summary: Deletes an existing player
      description: Deletes the player with the given ID
      operationId: deletePlayer
      parameters:
        ... # Pareil que le GET
      responses:
        '200':
          ... # Pareil que le GET
        '400':
          description: Invalid ID supplied
        '405':
          description: |-
            Cannot delete the player
            (playercards with it exists)
```

- Possible d'avoir plusieurs routes pour une URI
- C'est même une bonne pratique (URI = objet, verbes = actions sur cet objet)

Définir les DTO

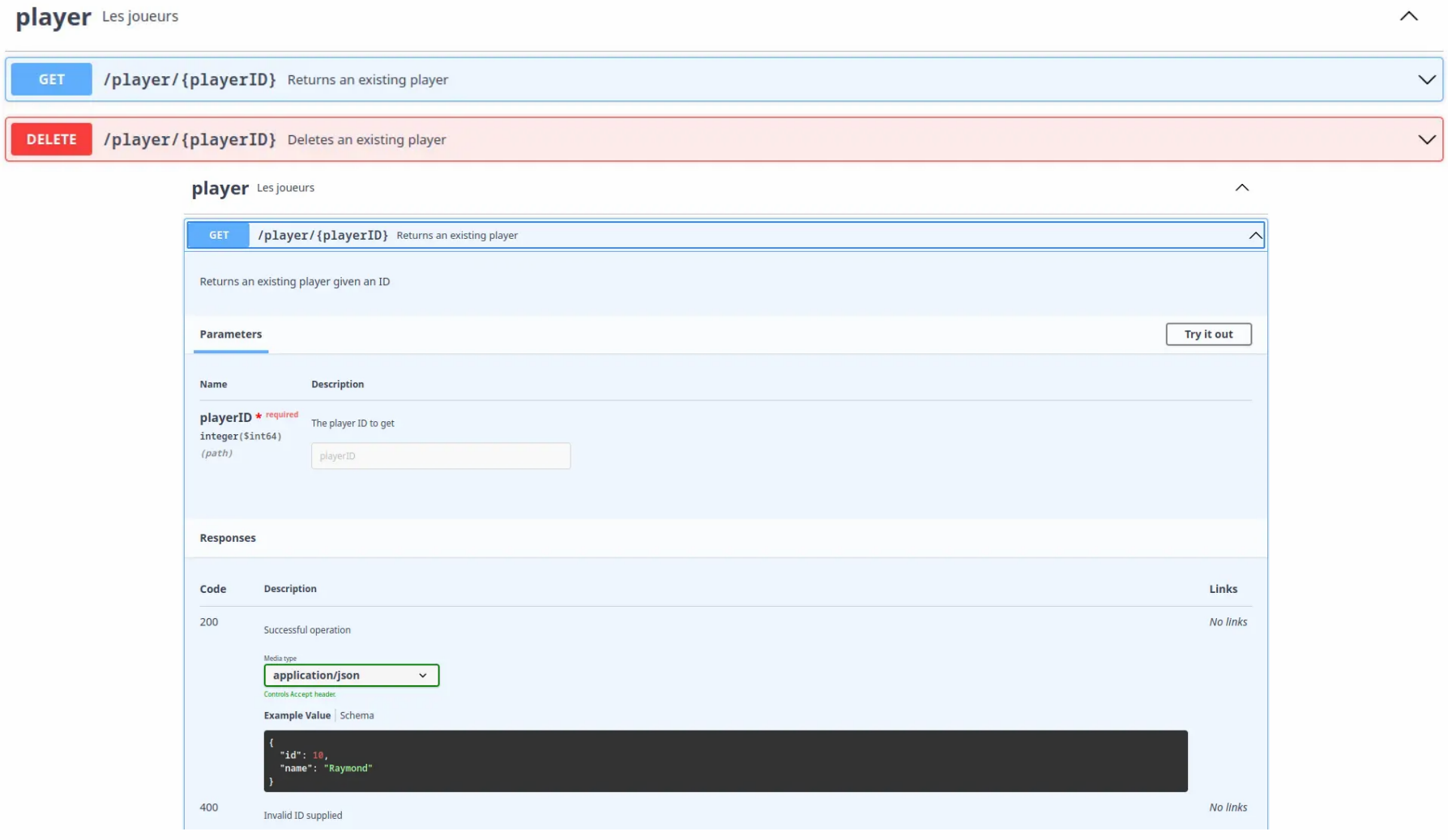
```yaml
openapi: 3.0.3
...
paths:
...
components:
  schemas:
    PlayerDTO:
      type: object
      properties:
        id:
          type: integer
          format: int64
          example: 10
        name:
          type: string
          example: Raymond
```

- Vous y faites donc référence dans le routes ainsi :

```yaml
schema:
  $ref: '#/components/schemas/PlayerDTO'
```

Concevoir son API REST avec OpenAPI

Résultats avec Swagger Editor



Concevoir son API REST avec OpenAPI

Question

Peut-on définir une route `GET /player/{playerID}` et une autre `GET /player/{nom}` ?

Concevoir son API REST avec OpenAPI

Une route retournant une liste

```yaml
/players/{playerName}:
  get:
    ...
    parameters:
      - name: playerName
        in: path
        description: The name of the players to get
        required: true
        schema:
          type: string
    responses:
      '200':
        description: Successful operation
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/ArrayPlayerDTO'
      ...
...
components:
  schemas:
    ...
    ArrayPlayerDTO:
      type: array
      items:
        $ref: '#/components/schemas/PlayerDTO'
```

Concevoir son API REST avec OpenAPI

Requête POST

- Données dans le body de la requête HTTP
- Un DTO spécifique pour le post (sans l'ID)

```yaml
/player: # pas de paramètre dans l'URI
  post:
    tags:
      - player
    summary: Adds a new player
    description: Adds a new player
    operationId: addPlayer
    requestBody: # Le contenu du body
      content: # Peut accepter JSON et XML
        application/json: # un PlayerNoIdDTO en JSON
          schema:
            $ref: '#/components/schemas/PlayerNoIdDTO'
      required: true
    responses:
      '200':
        description: Successful operation
        content: # On retourne l'objet crée avec son ID
          application/json:
            schema:
              $ref: '#/components/schemas/PlayerDTO'
      '405':
        description: Invalid input
```

```yaml
components:
  schemas:
    ...
    PlayerNoIdDTO:
      type: object
      properties:
        name:
          type: string
          example: Raymond
```

Concevoir son API REST avec OpenAPI

Requête PATCH

- Très similaire au POST
- Données dans le body de la requête HTTP

```yaml
/player: # pas de paramètre dans l'URI
  ...
  patch:
    tags:
      - player
    summary: Modifies an existing player
    description: Modifies an existing player
    operationId: patchPlayer
    requestBody:
      content:
        application/json:
          schema: # Cette fois un PlayerDTO car ID nécessaire
          # On modifie les attributs que l'on veut
            $ref: '#/components/schemas/PlayerDTO'
      required: true
    responses:
      '200':
        description: Successful operation
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/PlayerDTO'
      '405':
        description: Invalid input
```

Concevoir son API REST avec OpenAPI

Comment choisir les routes à définir ?

- Jamais au hasard
- Lire les spécifications pour comprendre les fonctionnalités / besoins
- Deux types de requêtes
  - CRUD
  - Orientée application
- **Route CRUD**: Create-Read-Update-Delete
  - Approche naïve : routes CRUD pour tous les objets
  - Simple / Générable par des outils (ex. Jhipser)
  - Pas nécessaire / pas judicieux / exhaustif
- **Route Orientée Application**
  - Design des routes en fonction des besoins du front-end
  - Exemple : la route `/albumsSummary`. Pas CRUD mais utile et optimisée pour obtenir les données nécessaires à l'affichage en résumé des albums telle que le demanderait un front-end
  - Complexifie le back-end car routes spécifiques
  - Permet d'optimiser les données (DTO) retournées

REST : service

```
@RestController
@RequestMapping("api/public/v1/hello")
public class HelloControllerV1 {
    private final Set<String> txts;
    private User user;

    public HelloControllerV1() {
        super();
        txts = new HashSet<>();
        txts.add("foo");
        txts.add("bar");
        user = new User("Foo", "here", "1", "p1");
    }
//...
}
```

- Pas terrible comme implémentation : les données (`txts` et `user`) sont stockées directement dans le contrôleur
- Séparation des préoccupations : chacun sa logique. Le contrôleur gère les routes, et un service va gérer les données

REST : service

- Un service est généralement une classe qui est dédiée à rendre certains types de ... services (accès données, etc.)
- En Spring c'est une classe qu'il faut annoter avec `@Service`
- On peut alors l'ajouter en paramètre des contrôleurs. Le service sera automatique créé (une fois) et injecté (patrons de conception : single instance, injection de dépendances)
- Le contrôleur utilise alors ce service pour manipuler les données et se focaliser sur les routes REST

```java
// To put in a package 'service'
@Getter
@Setter
@Service
public class DataService {
    private final Set<String> txts;
    private User user;

    public DataService() {
        super();
        txts = new HashSet<>();
        txts.add("foo");
        txts.add("bar");
        user = new User("Foo", "here", "1", "p1");
    }
}
```

```java
@RestController
@RequestMapping("api/public/v3/hello")
public class HelloControllerV3 {
    private final DataService dataService;

    public HelloControllerV3(final DataService dataService) {
        super();
        this.dataService = dataService;
    }

    @PatchMapping(path = "user",
    consumes = MediaType.APPLICATION_JSON_VALUE)
    public void patchUser(@RequestBody final UserDTO patchedUser) {
        patchedUser.patch(dataService.getUser());
    }
}
```

REST : gestion des exceptions

Exemple un peu grossier (mauvaise pratique) et faillible

```
@PutMapping(path = "user", consumes = MediaType.APPLICATION_JSON_VALUE)
public void replaceUser(@RequestBody final User patchedUser) {
  if(patchedUser.getId().equals(user.getId())) {
    user = patchedUser;
  } else { // we cannot change the id of the user
    // Is that a good practice? (no...)
    throw new IllegalArgumentException(user.toString());
  }
}
```

Le back-end REST transforme les exceptions en réponse HTTP avec le code 500

Résulat

```
curl -X PUT "http://localhost:8080/api/public/v1/hello/user" "Content-Type: application/json" -d '{ "name": "aa", "id":"10", "address":"there"}'

HTTP/1.1 500
{"timestamp":"2022-02-10T15:38:28.157+00:00","status":500,"error":"Internal Server Error","message":"User(name=aa, address=there, id=10, pwd=null)","path":"/api/public/v3/hello/user"}
```

Et dans la console d'IntelliJ:

```
java.lang.IllegalArgumentException: User(name=Foo, address=here, id=1)
    at fr.insarennes.demo.restcontrollers.HelloControllerV1.replaceUser(HelloControllerV1.java:59)
```

Attention à la fuite de données : Regardez le message retourné par *curl*

L'inclusion des messages se désactive dans `application.properties` en supprimant la ligne `server.error.include-message=always`

REST : gestion des exceptions

Bonne pratique

- Utilisez les codes de retour HTTP
- Obtenir une erreur 500 n'est jamais bon signe (erreur dans le back-end non gérée)

```
@PutMapping(path = "user", consumes = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<String> replaceUser(@RequestBody final User patchedUser) {
  if(patchedUser.getId().equals(dataService.getUser().getId())) {
    dataService.setUser(patchedUser);
    return ResponseEntity.ok().build();
  }
  throw new ResponseStatusException(HttpStatus.BAD_REQUEST, "The ID is not the same");
}
```

Résultat :

```
curl -X PUT "http://localhost:8080/api/public/v3/hello/user" "Content-Type: application/json" -d '{ "name": "aa", "id":"10", "address":"there"}' -i

HTTP/1.1 400
The ID is not the same
```

Plus propre, plus explicite. Mais attention à ne pas formuler des messages qui fourniraient des données sensibles (faire planter des back-ends est un vecteur d'attaque)

Communication serveur-BD

Le mapping objet <-> relationnel (Object-Relational Mapping, ORM)

Les *Repository*

- Ce sont des services spécifiques au stockage de données
- En Spring, deux principaux :
  - `CrudRepository`
  - `JpaRepository` (qui étend `CrudRepository`)
- On ne stocke pas à la main les données
  - On utilise une base de données (simple, Crud ; ou relationnelle, Jpa)
  - Optimisation du stockage
  - Facilité d'utilisation (gestion des clés uniques, etc.)
  - Requêtage
  - BD localisée sur un autre serveur

CrudRepository

Exemple :

```
@Repository
public interface FooCrudRepository extends CrudRepository<Foo, Long> {
}
```

- Définit un *repository* CRUD qui stockera des objets du type `Foo`
- Chaque objet `Foo` sera identifié de manière unique avec un clé du type `Long`

- Questions :
  - Différence entre `int` et `long` ?
  - Différence entre `long` et `Long` ?
  - Pourquoi un clé du type `long` plutôt que `int` ?

CrudRepository

La définition de `Foo` :

```
@Data
public class Foo {
  private long id;
  private String txt;
}
```

Utilisation du repository dans le service de `Foo` :

```
@Service
public class FooService {
  @Autowired
  private FooCrudRepository fooRepo;

  public Foo createNewFoo() {
    Foo foo = new Foo();
    foo.setTxt("hello");
    // sauvegarde dans le repo
    fooRepo.save(foo);

    return foo;
}}
```

Méthodes de base d'un *repository* :

```
// Enregistre un objet dans le repo.
// Le repo attribut une clé unique
// (Long) a cet objet
fooRepo.save(foo);

// Supprime le todo
fooRepo.delete(foo);

// Retourne l'objet correspond à
// la clé donnée. Attention retourne
// un Optional
fooRepo.findById(id);

// Retourne tous les objets stockés
// dans le repo
Iterable<Foo> it = fooRepo.findAll();
```

Attention : ce code ne fonctionne pas encore ! Cf. slide ORM concernant le repository

CrudRepository

Mais comment dire à Spring que l'attribut `id` est la clé unique à définir et utiliser dans le repository ?

```
@Data
public class Foo {
  private long id;
  private String txt;
}
```

En effet, de base le back-end ne sait pas quel attribut pour la clé unique utilisé lors du `save`.

```
fooRepo.save(foo);

fooRepo.findById(id);
```

Solution : utiliser un **ORM**

ORM -- Object-Relational Mapping

- A Web app usually has a database
- **Problems**:
  - database => relational schema (e.g. tables)
  - back-end => object-oriented (e.g. classes, objects)

=> Need automatic bindings between databases and server-side apps
=> Do not want to design and maintain both the database and the server-side app without facilities
=> Object-oriented <–> relational schema is not a simple translation

ORM -- Object-Relational Mapping

- To overcome these issues (see previous slide): **ORM – Object-Relational Mapping**
- **Goal**: Integrates databases with objects
- Converting objects' attributes into groups of values for storage in the database
- Converting database values back to object attributes upon retrieval
- ORM does not imply Web app. Can have a desktop app with a database
- **Why is it complex?**
  - Inheritance does not exist in relational databases
  - Primary key does not exist in OOP
  - Tables use foreign key, objects use references

ORM -- Object-Relational Mapping

Multiple ORM libraries exist

- Famous ones:
  - **JPA** (Jakarta Persistance API), aka. **Jakarta**. The one we will use within Spring Boot
  - **Spring Data**. Spring Boot can work without Jakarta with its own annotations. But Jakarta is almost a ORM standard multiples frameworks comply to.

ORM -- Object-Relational Mapping

How to define the ORM of a back-end?

Two ways: Java annotations or XML files. We will use Java annotations

Java annotations

- Examples: @Override, @Test. Can tag classes, attributes, etc. (depends on their definition)
- In our case (with JPA):
  @Id
  @GeneratedValue
  @Entity
  @ManyToOne @JoinColumn(nullable = false)
  @JoinColumn(name="ENSEIGNANT_ID", nullable = false)
  @OneToMany(mappedBy = "agenda", cascade = CascadeType.PERSIST, fetch= FetchType.LAZY)
  @NamedQueries
  @NamedQuery
  @MappedSuperclass

A processor will analyse the JPA annotations of the program to build the database and the object-relational mapping. The annotation details the database schema and how the mapping will work. This is automatic.

ORM -- Object-Relational Mapping

@Entity

If instances of a class must be serialized in the database, the class must be tagged with `@Entity`

```
import jakarta.persistence.Entity;

@Data
@Entity
public class Foo {
  private long id;
  private String txt;
}
```

But an entity must have a primary key

ORM -- Object-Relational Mapping

@Id

Defines the attribute as the primary key

@GeneratedValue

The value of the attribute is automatically initialized and incremented

Note that annotations can be put on getters or attributes

```
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.Id;
@Data
@Entity
public class Foo {
  @Id
  @GeneratedValue
  private long id;
  private String txt;
}
```

With these annotations, the repository example now works

ORM -- Object-Relational Mapping

- New example: `Foo` has `Bars`. `Bar` knows is `Foo`

```
@Data
@Entity
public class Foo {
  @Id
  @GeneratedValue
  private long id;
  private String txt;
  private List<Bar> bars;
}
```

```
@Data
@Entity
public class Bar {
  @Id
  @GeneratedValue
  private long id;
  private Foo foo;
}
```

- Using this example the back-end will crash during its execution
- Why? Because it does not know how to transform the Java reference `List<Bar> bars` and its opposite `private Foo foo` into a relational schema
- Remind that you must explain Spring how to generate the relation schema of the underlying database
  - Define the primary / foreign keys
  - Define the tables

ORM -- Object-Relational Mapping

```java
@Data
@Entity
public class Foo {
  @Id
  @GeneratedValue
  private long id;
  private String txt;
  @OneToMany(mappedBy = "list")
  private List<Bar> bars;
}
```

```java
@Data
@Entity
public class Bar {
  @Id
  @GeneratedValue
  private long id;
  @ManyToOne
  private Foo foo;
}
```

- `@OneToMany`: One `Foo` has several `Bars`
- `@ManyToOne`: multiple `Foos` can refer one same `Bar`
- Database script:

```sql
-- The todo table now has a list_id column
create table todo (..., list_id bigint, primary key (id));

-- Since a todo_list has a list of todo, no change in the table
create table todo_list (id bigint not null, ...,
   primary key (id));

-- But foreign key for each todo parts of the todo_list
alter table if exists todo add constraint ...
   foreign key (list_id) references todo_list;
```

ORM -- Object-Relational Mapping

```
@Data
@Entity
public class Foo {
  @Id
  @GeneratedValue
  private long id;
  private String txt;

  @OneToMany(mappedBy = "list",
      cascade = CascadeType.PERSIST,
      fetch = FetchType.LAZY)
  private List<Bar> bars;
}

...
```

Various parameters for `OneToMany`:

- `cascade` = what kind of operations applied on the object must be applied on children too? operations: *persist, remove, merge, refresh, all*
- `fetch`: strategy about how data are fetched `lazy`: data loaded on demand only `eager`: data always loaded

ORM -- Object-Relational Mapping

```
@Data
@Entity
public class Foo {
  @Id
  @GeneratedValue
  private long id;
  private String txt;
  @ManyToMany
  private List<Bar> bars;
}
```

```
@Data
@Entity
public class Bar {
  @Id
  @GeneratedValue
  private long id;
  @ManyToMany
  private List<Foo> foo;
}
```

`ManyToMany`:

- the source object has a collection of target objects
- the target object has a collection of source objects

One `Foo` has `Bars`
One `Bar` can be part of several `Foos`

ORM -- Object-Relational Mapping

```java
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
public class Player extends ModelElement {
}

@Entity
public class BaseballPlayer extends Player {
  private int totalHomeRuns;
}
```

Java => inheritance

No inheritance in relational algebra

Inheritance between entities: `@Inheritance`

Different strategies:

- `SINGLE_TABLE`: a single table for all the inheritance hierarchy (here, a unique table or both `Player`, `BaseballPlayer`)

`DTYPE` defines the type

All the attributes added to the table

```
CREATE TABLE PLAYER (ID NUMBER(10) NOT NULL, DTYPE VARCHAR(31), P_NAME VARCHAR NOT NULL, TOTALHOMERUNS NUMBER(10), PRIMARY KEY (ID))
```

ORM -- Object-Relational Mapping

```java
@Entity
// SINGLE_TABLE, TABLE_PER_CLASS, JOINED
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Player extends ModelElement {
}

@Entity
public class BaseballPlayer extends Player {
  private int totalHomeRuns;
}
```

Java => inheritance

No inheritance in relational algebra

Inheritance between entities: `@Inheritance`

Different strategies:

- `TABLE_PER_CLASS`: one table per class (columns duplicated)

```
CREATE TABLE PLAYER (ID NUMBER(10) NOT NULL, P_NAME VARCHAR NOT NULL,
                     PRIMARY KEY (ID))
CREATE TABLE BASEBALLPLAYER (ID NUMBER(10) NOT NULL,
  P_NAME VARCHAR NOT NULL, TOTALHOMERUNS NUMBER(10), PRIMARY KEY (ID))
```

- `JOINED`: one table per class but columns factorised (foreign key)

```
CREATE TABLE PLAYER (ID NUMBER(10) NOT NULL, DTYPE VARCHAR(31),
                            P_NAME VARCHAR NOT NULL, PRIMARY KEY (ID))
CREATE TABLE BASEBALLPLAYER (ID NUMBER(10) NOT NULL,
                            TOTALHOMERUNS NUMBER(10), PRIMARY KEY (ID))
ALTER TABLE BASEBALLPLAYER ADD CONSTRAINT FK_BASEBALLPLAYER_ID
                              FOREIGN KEY (ID) REFERENCES PLAYER (ID)
```

ORM + Repository => Query

- We saw repositories store data
- Defined in Spring using a simple interface
- Can add methods in those interfaces and annotate them with SQL-like **queries**

```
@Repository
public interface UserCrudRepository extends CrudRepository<User, Long> {
    @Query("select u from User u where u.name like %?1%")
    List<Todo> findUserNameContainsTxt(final String nameTxt);
}
```

- This query returns all the users whose name contains the give string
- Can use such method in services

```
@Service
public class UserService {
    @Autowired
    private UserCrudRepository repository;
  //...
  public List<User> findUserByNameFragment(final String nameFragment) {
      return repository.findUserNameContainsTxt(nameFragment);
  }
}
```

REST : tester les routes REST

Principes

- On test de manière unitaire chaque route REST (avec JUnit, et d'autres librairies)
- Le lancement d'un test, lance le back-end en mode test
- Que testons-nous ?
  - Le **bon fonctionnement** (test nominal).
    Formats d'entrée, paramètres, résultats, réponse, etc.
  - La **résilience aux requêtes mal formées** (intentionnellement ou non)
    - Mauvais format des données, mauvaises structures, mauvais paramètres (faire crasher le back-end peut être un vecteur d'attaque)
    - Injections (SQL)
    - Accès à des routes REST nécessitant une authentification
    - D'autres idées ? (cf cours vulnérabilité)
  - **Performance et montée en charge**
    - Est-ce que mes back-ends tiennent la charge d'un million d'utilisateur en même temps ?
    - Est-ce que telle route REST répond en un temps acceptable ?

REST : tester les routes REST

Créons une classe `AnimalControllerTest` dans le dossier `src/test/java/demo`

```
// dans src/test/java/demo
@SpringBootTest
@AutoConfigureMockMvc
class AnimalControllerTest {
    @Autowired
    private MockMvc mvc;

    @Autowired
    private AnimalService animalService;
}
```

Pour rappel :

```
@RestController
@RequestMapping("api/public/v1/animal")
public class AnimalController {
    private final AnimalService animalService;
    public AnimalController(AnimalService service) {
        animalService = service;
    }
  //...
}
```

```
@Getter @Service
public class AnimalService {
    private final List<Animal> animals;
    public AnimalService() {
        animals = new ArrayList<>();
        animals.add(new Cat(1, "foo"));
        animals.add(new Dog(2, "bar"));
    }
}
```

- `@SpringBootTest` lance le back-end en mode test
- `@AutoConfigureMockMvc` configure automatique les ressources/contrôleurs REST
- `@Autowired` = injection de dépendances
- L'attribut `animalService` est notre service qui contient les données des animaux. Annoté par `@Autowired` il est injecté (créé et assigné) automatiquement
- L'attribut `MockMvc mvc` va nous permettre d'exécuter et de tester nos requête REST. Il est également injecté

REST : tester les routes REST

On ajoute un test dans notre classe `AnimalControllerTest`

```
@Test
void getAll() throws Exception {
  // performing the query
  mvc.perform(get("/api/public/v1/animal/all"))
    // checking the status code
    .andExpect(status().isOk())
    // checking the returned data format
    .andExpect(content()
       .contentTypeCompatibleWith(MediaType.APPLICATION_JSON))
    // printing in the console the returned data
    .andDo(MockMvcResultHandlers.print())
    // checking the JSON structure of the returned data
    .andExpect(jsonPath("$", hasSize(2)))
    .andExpect(jsonPath("$[0].name", equalTo("foo")))
    .andExpect(jsonPath("$[1].name", equalTo("bar")))
    .andExpect(jsonPath("$[0].age", equalTo(1)))
    .andExpect(jsonPath("$[1].age", equalTo(2)));
}
```

Ce test vise la route suivante de `AnimalController` :

```
@GetMapping(path = "all", produces = MediaType.APPLICATION_JSON_VALUE)
public List<Animal> getAll() {
  return animalService.getAnimals();
}
```

- `mvc.perform` exécute la requête REST donnée
- `get("/api/public/v1/animal/all")` définit la route REST (`MockMvcRequestBuilders.get(...)` en fait)
- les appels `andExpect` sont les assertions
- `.andExpect(status().isOk())` vérifie le code de retour
- `.andExpect(content()...)` vérifie que les données retournées sont en JSON
- `.andDo(MockMvcResultHandlers.print())` affiche en console les données retournées
- `.andExpect(jsonPath(...))` vérifie la structure du JSON reçu

REST : tester les routes REST

```
@Test
void getAll() throws Exception {
  // performing the query
  mvc.perform(get("/api/public/v1/animal/all"))
    // checking the status code
    .andExpect(status().isOk())
    // checking the returned data format
    .andExpect(content()
       .contentTypeCompatibleWith(MediaType.APPLICATION_JSON))
    // printing in the console the returned data
    .andDo(MockMvcResultHandlers.print())
    // checking the JSON structure of the returned data
    .andExpect(jsonPath("$", hasSize(2)))
    .andExpect(jsonPath("$[0].name", equalTo("foo")))
    .andExpect(jsonPath("$[1].name", equalTo("bar")))
    .andExpect(jsonPath("$[0].type", equalTo("cat")))
    .andExpect(jsonPath("$[1].type", equalTo("dog")))
    .andExpect(jsonPath("$[0].age", is(1)))
    .andExpect(jsonPath("$[1].age", is(2)));
}
```

JSON reçu :

```
[
  {"type":"cat", "age":1, "name":"foo"},
  {"type":"dog", "age":2, "name":"bar"}
]
```

- `jsonPath` a deux paramètres : l'élément du JSON à vérifier ; l'assertion à exécuter sur cet élément
- `"$"` : c'est la racine du JSON
- Ici le JSON est un tableau. On vérifier sa taille : `.andExpect(jsonPath("$", hasSize(2)))`
- Pour prendre un élément du tableau JSON : `$[0]`, `$[1]`, etc.
- Pour prendre un élément du JSON : `$[0].name`, `$[1].age`, etc. Si la racine n'avait pas été un tableau, on aurait pu écrire : `$.monAttr`
- Spring n'utilise pas uniquement *JUnit* pour les assertions mais aussi *Hamcrest* (`hasSize`, `equalTo`, `is`, etc.)

REST : tester les routes REST

- `hasSize`, `equalTo`, `is` sont des méthodes statiques de la classe `Matchers` de `org.hamcrest`
- Quelle différence en `equalTo` et `is` ?
- La liste des assertions Hamcrest : http://hamcrest.org/JavaHamcrest/javadoc/2.2/org/hamcrest/Matchers.html
- Vous pouvez écrire `Matchers.hasSize(...)` par exemple. Sinon il faut dire à IntelliJ de faire un *import static* : `import static org.hamcrest.Matchers.equalTo;`
- De même, `jsonPath` est une méthode statique de la classe `MockMvcRequestBuilders`

REST : tester les routes REST

On ajoute un autre test dans notre classe `AnimalControllerTest`

```
@ParameterizedTest
@MethodSource("animalsProvider")
void postAnimal(final Animal animal) throws Exception {
  mvc.perform(
      post("/api/public/v1/animal")
        .contentType(MediaType.APPLICATION_JSON)
        .content(new ObjectMapper().writeValueAsString(animal))
    )
    .andDo(MockMvcResultHandlers.print())
    .andExpect(status().isOk())
    // check that the returned body is empty
    .andExpect(content().string(""));

  assertEquals(animalService.getAnimals()
    .get(animalService.getAnimals().size() - 1), animal);
}

static Stream<Animal> animalsProvider() {
  return Stream.of(
    new Cat(20, "c"),
    new Dog(30, "d")
  );
}
```

Ce test vise la route suivante de `AnimalController` :

```
@PostMapping(path = "",
  consumes = {MediaType.APPLICATION_JSON_VALUE,
              MediaType.APPLICATION_XML_VALUE})
public void newAnimal(@RequestBody final Animal animal) {
  animalService.getAnimals().add(animal);
}
```

- On veut tester que l'ajout de tous les types d'animaux fonctionne
- Je ne vais pas écrire un test pour chaque type car la structure des tests seront toutes identiques
- On écrit alors un test paramétré : un même test prend en paramètre une source de données (`Stream` de `Animal`)
- Cette source de données est la méthode `animalsProvider` qui fournit un `Cat` et un `Dog`
- Le test sera donc exécuté deux fois : une fois pour chaque donnée d'entrée
- `post(...).contentType(...).content(...)` réalise un POST en définissant le format du body, ainsi que les données JSON du body
- `new ObjectMapper().writeValueAsString` marshalle l'objet donné pour produire du JSON
- le `assertEquals` vérifie que le service a le nouvel animal

REST : tester les routes REST

Oui mais : nous n'avons pas tout testé sur cette route (format entrée)

```
@ParameterizedTest
@MethodSource("postAnimalProvider")
void postAnimal(Animal animal, String type, ObjectMapper marshaller) throws Exception {
  mvc.perform(
      post("/api/public/v1/animal")
        .contentType(type)
        .content(marshaller.writeValueAsString(animal))
    )
    .andDo(MockMvcResultHandlers.print())
    .andExpect(status().isOk())
    // check that the returned body is empty
    .andExpect(content().string(""));

  assertEquals(animalService.getAnimals()
    .get(animalService.getAnimals().size() - 1), animal);
}

static Stream<Arguments> postAnimalProvider() {
  return animalsProvider()
    .map(animal ->
      Stream.of(
        new Pair<>(MediaType.APPLICATION_JSON_VALUE, new ObjectMapper()),
        new Pair<>(MediaType.APPLICATION_XML_VALUE, new XmlMapper())
      ).map(type -> Arguments.of(animal, type.first(), type.second())))
    .flatMap(s -> s);
}


@PostMapping(path = "",
  consumes = {MediaType.APPLICATION_JSON_VALUE,
              MediaType.APPLICATION_XML_VALUE})
public void newAnimal(@RequestBody final Animal animal) {
  animalService.getAnimals().add(animal);
}
```

- Ouch la méthode `postAnimalProvider`. Elle fait le produit cartesien des animaux et des formats
- Le résultat de l'exécution du test sera donc 4 tests :

  `[cat, JSON] [cat, XML], [dog, JSON], [dog, XML]`

- Notre méthode de test à maintenant trois paramètres, utilisés dans le test
  - L'animal
  - Le format d'entrée (`type`)
  - Le marshaller (`ObjectMapper`) correspondant au format d'entrée, car XML et JSON n'utilisent pas le même marshaller
- `Arguments.of` définit la liste des arguments à fournir au test (ici nous avons donc trois arguments)

REST : tester les routes REST

Test de résilience

- La sécurité des applications Web est primordiale (fuites de données, usurpation d'identité, mise à mal de services critiques [centrales, transport, etc.], dégradation de l'image de l'entreprise, etc.)
- Log4Shell (CVE-2021-44228) ?

```
@Test
void postBadAnimalNoType() throws Exception {
  mvc.perform(
      post("/api/public/v1/animal")
        .contentType(MediaType.APPLICATION_JSON)
        .content("""
          { "age":2,"name":"bar" }""")
    )
    .andExpect(status().isBadRequest())
    .andExpect(content().string(""));
}
```

- Il existe des techniques spécifiques de test : fuzzy testing (5INFO)
- Au fait, vous connaissez `""" ... """` en Java ? https://openjdk.java.net/jeps/378

REST : tester les routes REST

Analyse de vulnerabilités

- https://owasp.org/www-project-dependency-check/
  - Ajouté aux projets de démo et du TP
- Il existe des outils qui s'intègre dans vos projets front-end et back-end pour analyser le code
- *SpotBug*, *ErrorProne*, *ESLint*, etc., trouvent des bugs potentiels, des mauvaises pratiques
- *dependency-check* vérifie la présence de certains CVE dans votre application
  - Dans IntelliJ, panneau Maven -> Plugins -> dependency-check -> dependency-check:check
- Exemple de CVS dans Jackson : https://nvd.nist.gov/vuln/detail/CVE-2020-36180
- En front-end : `npm audit`

Authentification / Sécurisation des routes REST

- **Indispensable** pour les applications industrielles
- Cf. cours hygiène informatique + vulnérabilité (Cloud/Sécurité)
- **Buts**
  - Tous les utilisateurs n'ont **pas les même droits** (admin, anonyme, utilisateur identifié, utilisateur identifié avec privilèges [exemple : Netflix en HD ;)])
  - Donc **protection des données** (accès à certains services de l'appli, données personnelles)
  - **Personnaliser** les utilisateurs connectés (préférences, derniers films vus, etc.)
- Pour l'instant, les routes REST utilisées étaient publiques (je les ai configurées pour que des utilisateurs non authentifiés puissent les utiliser, sans limite)
  - Attention : c'est une démo
  - **Problème des API publiques** : déni de service, utilisation abusive, etc. (contre-mesure : les sessions)
- Aussi, nous travaillons en HTTP et non en HTTPS. Évidemment dans la vraie vie **HTTPS partout**. Pourquoi ?
- Avec *Spring*, nous utiliserons *Spring Security*

Authentification / Sécurisation des routes REST

Configuration actuelle de la sécurité du back-end Spring

- `antMatchers("/api/public/**").permitAll()` Toutes les routes qui suivent ce format sont accessibles par tout le monde
- `.anyRequest().authenticated()` Et toutes les autres routes nécessiterons une authentification
- `.csrf().disable()` Pour la démo, j'ai désactivé la sécurité contre l'attaque *Cross-site request forgery*. C'est quoi ? (Ne pas confondre avec Cross-site Scripting -- XSS)
  https://docs.spring.io/spring-security/reference/features/exploits/csrf.html
  https://owasp.org/www-community/attacks/csrf

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
  //...

    @Override
    protected void configure(final HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
//            .antMatchers("/api/admin/**").hasRole("ADMIN")
            .antMatchers("/api/public/**").permitAll()
            .anyRequest().authenticated()
            .csrf().disable();
    }
}
```

Authentification / Sécurisation des routes REST

Mécanismes d'authentification

Mauvaise pratique

- Fabriquer soit même son système d'authentification. Nous pensons rarement à toutes les attaques possibles. Ne pas réinventer la roue

Bonne pratique

- Utiliser une système d'authentification éprouvé
  - celui des librairies industrielles (Spring Security)
  - OpenID
  - OAuth 2.0 : délégation d'autorisation (et non système d'authentification). Exemples ?
  - Shibboleth (fédération d'autorités d'authentification, donc délégation d'autorisation). Exemples ?
  - CAS (Central Authentication Service)

Authentification / Sécurisation des routes REST

Mécanismes d'authentification avec Spring Security

- Support de OAuth 2.0, etc.
- Nous utiliserons le gestionnaire d'authentification en mémoire (pas de base de données des utilisateurs)

```
@RestController
@RequestMapping("api/public/user")
@AllArgsConstructor
public class PublicUserController {
    private final UserService userService;

    @PostMapping(value = "new", consumes = MediaType.APPLICATION_JSON_VALUE)
    public void newAccount(@RequestBody final UserDTO user) {
        try {
            userService.newAccount(user.login(), user.pwd());
        }catch(final IllegalArgumentException ex) {
            throw new ResponseStatusException(HttpStatus.BAD_REQUEST, "Not possible");
        }
    }

    @PostMapping(value = "login", consumes = MediaType.APPLICATION_JSON_VALUE)
    public void login(@RequestBody final UserDTO user) {
        try {
            final boolean logged = userService.login(user.login(), user.pwd());

            if(logged) {
                throw new ResponseStatusException(HttpStatus.BAD_REQUEST, "Already logged in. Log out first");
            }
        }catch(final ServletException ex) {
            throw new ResponseStatusException(HttpStatus.BAD_REQUEST, "Not possible to log in");
        }
    }
}
```

- 2 routes REST publiques
- `POST 'api/public/user/new'` pour créer un compte
- `POST 'api/public/user/login'` pour s'identifier
- `UserDTO` est juste un tuple identifiant/mot de passe
- Attention à la gestion des mots de passe (HTTPS, à mettre dans le body d'un POST, stockage du hash côté back-end)
- Pour le hash des mots de passe, nous utiliserons *bcrypt*
- `UserService` est un service qui gère les utilisateurs

Authentification / Sécurisation des routes REST

Création d'un utilisateur

```
curl -X POST "http://localhost:8080/api/public/user/new" -H  "Content-Type: application/json" -d '{"login": "user", "pwd":"user"}' -i

HTTP/1.1 200
```

Qui peut voir le payload (les données du body) avec HTTPS ?

Authentification d'un utilisateur

```
curl -X POST "http://localhost:8080/api/public/user/login" -H  "Content-Type: application/json" -d '{"login": "user", "pwd":"user"}' -i

HTTP/1.1 200
Set-Cookie: JSESSIONID=6979E022B610B2FF125BEA740939E6B9
```

Qu'est ce que `Set-Cookie` ? Et `JSESSIONID` ?

Authentification / Sécurisation des routes REST

Cookie et session

- Un cookie (protocole HTTP) stocke des informations pour : faire des choses douteuses (...), stocker des informations de sessions, des préférences, des dates de connexions, etc.
- `Set-Cookie` nous dit qu'un cookie a été crée et transmit du back-end au front-end via la réponse HTTP
- `JSESSIONID` est notre identifiant confidentiel de session. Une session associe un utilisateur avec un identifiant de session (et des dates) pour, dans notre cas, autoriser l'utilisateur à utiliser des routes privées : on se connecte une fois, puis on utilise cet identifiant en guise de preuve d'identité
- Est-ce que l'on peut se faire voler son identifiant de session stocké dans un cookie ? https://stackoverflow.com/questions/4150153/what-prevents-httpsessions-id-from-being-stolen
- Sur certains site Web le `jsessionid` est directement dans l'URL. Une bonne idée à votre avis ?

Authentification / Sécurisation des routes REST

Utilisation de routes privées avec identifiant de session

```java
@RestController
@RequestMapping("api/private/user")
@AllArgsConstructor
public class PrivateUserController {
    private final UserService userService;

    @GetMapping()
    public String hello(final Principal user) {
        return user.getUsername();
    }

    @PostMapping("out")
    public void logout() {
        try {
            userService.logout();
        }catch(final ServletException | IllegalStateException ex) {
            throw new ResponseStatusException(HttpStatus.BAD_REQUEST,"Cannot log out");
        }
    }
}
```

- 2 routes REST privées
- `GET 'api/private/user'` retourne le login de la personne connecté
- `POST 'api/public/user/login/out'` pour de déconnecter
- Le paramètre `Principal` est injecté. Il correspond à l'utilisateur connecté

On utilise le `JSESSIONID` des requêtes précédentes en envoyant un cookie avec cet ID :

```
curl --cookie 'JSESSIONID=6979E022B610B2FF125BEA740939E6B9' -X GET "http://localhost:8080/api/private/user" -i
```

Résultat : `HTTP/1.1 200 user`
Sans l'ID : `HTTP/1.1 403`

Authentification / Sécurisation des routes REST

Questions :

- Qu'est ce qui se passe, en terme de session, si j'appelle `POST 'api/public/user/login'` deux fois de suite ?

- Qu'est ce qui se passe, en terme de réponse HTTP, si j'appelle `` `POST 'api/public/user/login/out' `` sans ID de session ?

Authentification / Sécurisation des routes REST

Slide un peu compliqué qui montre comment le service des utilisateurs fonctionne avec Spring

```java
@Service
@AllArgsConstructor
public class UserService {
    private final PasswordEncoder passwordEncoder;
    private final UserDetailsManager userDetailsManager;
    private final HttpServletRequest request;

    public void newAccount(final String login, final String pwd) {
        if(userDetailsManager.userExists(login)) {
            throw new IllegalArgumentException("Not possible");
        }

        final UserDetails user = new User(login, passwordEncoder.encode(pwd),
            Collections.singletonList(new SimpleGrantedAuthority("ROLE_USER")));
        userDetailsManager.createUser(user);
    }

    public boolean login(final String login, final String pwd) throws ServletException {
        final HttpSession session = request.getSession(false);

        if(session == null) {
            request.getSession(true);
            request.login(login, pwd);
            return true;
        }
        return false;
    }

    public boolean logout() throws ServletException {
        final HttpSession session = request.getSession(false);

        if(session == null) {
            return false;
        }
        request.logout();
        return true;
    }
}
```

- Les trois attributs du service sont injectés (fourni par `SecurityConfig`)
- `PasswordEncoder` permet d'encoder les mots de passe
- `UserDetailsManager` stocke les utilisateurs en mémoire
- `HttpServletRequest` permet de gérer les sessions
- On stocke le hash du mot de passe

Une brève introduction à XML, JSON et YAML

XML

- eXtensible Markup Language
- Description and exchange (meta-)language for structured documents
  - XML is a language
  - XML permits the definition of dedicated XML languages (so a meta-language). Will see later how
- From W3C https://www.w3.org/XML/

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0">
    <groupId>web</groupId>
    <artifactId>tpREST</artifactId>
    <packaging>war</packaging>
    <version>1.0.0</version>
    <name>tpREST</name>

    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.glassfish.jersey</groupId>
                <artifactId>jersey-bom</artifactId>
                <version>${jersey.version}</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>
</project>
```

XML elements and attributes

- An XML element (becarefull: XML is case sensitive)
  `<name> content</name>`
  `<name attribute1='value' attribut2='value'> content </name>`

- An XML element tree
  - the **root** element: a single XML element that has children (tree structure)
  - each child is necessarily nested in its parent

- An XML attribute
  - pairs : `name="value"` or `name='value'`
  - Example: `<report language= 'FR' modifdate= '05-10-28'>... </report>`

- Empty element: `<elem></elem>` or the simplified writing: `<elem/>`

- Comment: `<!- - - ->`. Example: `<!- - this is a comment - ->`

XML well-formedness

Well-formed XML document => satisfies XML syntactic rules

Well-formed XML document

```
<project xmlns='maven.apache.org/POM/4.0.0'>
    <groupId>web</groupId>
    <artifactId>tpREST</artifactId>
    <packaging>war</packaging>
    <version>1.0.0</version>
    <name>tpREST</name>
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.glassfish.jersey</groupId>
                <artifactId>jersey-bom</artifactId>
                <version>${jersey.version}</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>
</project>
```

Not Well-formed

```
<project xmlns='maven.apache.org/POM/4.0.0'>
    <groupId>web</groupId>
    <artifactId>tpREST</artifactId>
    <packaging>war</packaging>
    <version>1.0.0</version>
    <name>tpREST</name>
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.glassfish.jersey
                <artifactId>jersey-bom</artifact>
                <version>${jersey.version}<version>
                <type>pom</type>
                <scope/>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>
</project>
```

To check a document:

`xmllint illformedDoc.xml`

XML Validity

- Valid XML document => well-formed + conforms to its schema definition
- Schema: document that defines an dedicated XML format / sub-language
- Examples: SVG, POM, XMLSchema, RSS, XSLT
- Can define an XML document without a schema

A POM document

```
<project xmlns="maven.apache.org/POM/4.0.0">
    <groupId>web</groupId>
    <artifactId>tpREST</artifactId>
    <packaging>war</packaging>
    <version>1.0.0</version>
    <name>tpREST</name>
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.glassfish.jersey</groupId>
                <artifactId>jersey-bom</artifactId>
                <version>${jersey.version}</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>
</project>
```

The schema of the XML-based POM language in XMLSchema

```
<xs:schema targetNamespace="maven.apache.org/POM/4.0.0">
  <xs:element name="project" type="Model">
    <xs:annotation>
      <xs:documentation source="version">3.0.0+</xs:documentation>
      <xs:documentation source="description">
        The <code>&lt;project&gt;</code> element is the root of the descriptor.
        The following table lists all of the possible child elements.
      </xs:documentation>
    </xs:annotation>
  </xs:element>
...
```

To check a document:

```
xmllint --schema maven-4.0.0.xsd validdoc.xml
```

XML Namespace

- One XML document can mix several XML languages
- To avoid conflicts (two languages sharing a same tag): namespace
- `xmlns`: the standard namespace: tags are used without prefix
- `xmlns:theprefixyouwant`: the tags prefixed with `theprefixyouwant` will identify one specific XML language

POM will be the main language

```
<project xmlns="maven.apache.org/POM/4.0.0">
    <groupId>web</groupId>
    <artifactId>tpREST</artifactId>
    <packaging>war</packaging>
    <version>1.0.0</version>
    <name>tpREST</name>
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.glassfish.jersey</groupId>
                <artifactId>jersey-bom</artifactId>
                <version>${jersey.version}</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>
</project>
```

xs is a prefix defined somewhere to identify the XMLSchema language
are tags part of XMLSchema

```
<xs:schema targetNamespace="maven.apache.org/POM/4.0.0">
  <xs:element name="project" type="Model">
    <xs:annotation>
      <xs:documentation source="version">3.0.0+</xs:documentation>
      <xs:documentation source="description">
        The 'project' element is the root of the descriptor.
      </xs:documentation>
    </xs:annotation>
  </xs:element>
...
```

XML: schema definition

- Valid XML document => well-formed + conforms to its schema definition
- **Schema: Definition of the 'vocabulary' and the structure of an XML document**
- Several languages to define schemas: DTD, XML Schema, RelaxNG
- Schema are not mandatory, i.e. can define an XML document with defining its structure

DTD: Document type definition
Example:

```
<!ELEMENT person (name,address*)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT address (#PCDATA)>
<!ATTLIST person idcard CDATA #REQUIRED>
```

With a valid XML document:

```
<person idcard="1843739">
    <name>John Doe</name>
    <address>Adress 1</address>
    <address>Adress 2</address>
</person>
```

More information:
http://www.w3schools.com/xml/xml_dtd_intro.asp
http://www.w3schools.com/xml/schema_intro.asphttp://r
20011203.html

XML: schema definition (XMLSchema)

XMLSchema

Example:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="www.w3.org/2001/XMLSchema">
    <xs:element name="person">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="name" type="xs:string"
                    maxOccurs="1"/>
                <xs:element name="address" type="xs:string"
                    minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
            <xs:attribute name="idcard" type="xs:string" use="required"/>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

With a valid XML document:

```
<person idcard="1843739">
    <name>John Doe</name>
    <address>Adress 1</address>
    <address>Adress 2</address>
</person>
```

```
xmllint --schema person.xsd person.xml
```

More information:
https://www.w3schools.com/xml/schema_intro.asp

JSON

- JavaScript Object Notation
- Pronounce Jay-zone (not jee-zon)
- Semi-structured attribute-value format
- No schema

Example:

```
{
    "idcard": 1843739,
    "name": "John Doe",
    "address": ["Adress 1", "Adress 2"],
    "phone": {
        "prefix": "+33",
        "number": "000000"
    },
    "siblings": null,
    "alive": false
}
```

See: https://www.w3schools.com/js/js_json_intro.asp

- Objects are defined between curly braces
- Contrary to XML: no element vs attribute
- An object contains a list of tuples (separated by `,`)
- **Tuple** = `"key": value`
- **Value**: array, string, nested object, boolean, number, null

YAML

- Yet Another Markup Language (YAML Ain't Markup Language): data-oriented language
- Semi-structured attribute-value format
- No schema
- The indentation defines the nested structure
- More features than JSON, such as comments

Example:

```yaml
persons: # List of persons
  - idcard: !!integer 1843739
  name: John Doe
  address:
    - Address 1
    - Address 2
  phone:
    prefix: !!string +33
    number: !!integer 000000
  siblings: null
  alive: false

  - idcard: !!integer 12212
...

---
```

- See: https://yaml.org
- Example: https://editor.swagger.io
- `!!integer` documents the type of the attribute (integer, string, float, etc.)
- No comma, option quote
- `---` separates YAML documents located in the same file

Point réseau

Au fait, vous savez ce qu'est ?:

- **localhost**
- **12.111.81.35**
- **fd10:bbbc:bcbb:ba01::2f:1f2c**
- **0.0.0.0** (https://fr.wikipedia.org/wiki/0.0.0.0)
- un **nom de domaine**
- un **DNS**
- **http**, **https**, **smtp**, **ftp**, **sftp**, **imap**, etc.

https://en.wikipedia.org/wiki/Lists_of_network_protocols

Front-end

Quelles sont les différentes techniques pour produire des front-ends

- **Application front-end indépendante**
  Exemples : *Angular*, *React*
  Le front-end est une application Web indépendante du back-end. Elle se connecte au back-end à l'aide de requêtes REST, WebSockets
  Angular : 4INFO

- **Rendu de front-end à base de templates** (template engine)
  Exemples : *php Symphony*, *Spring Boot* avec *Thymeleaf* (https://www.thymeleaf.org/)
  Le back-end produit les pages Web du front-end en utilisant des templates (des morceaux de pages Web à compléter pendant l'exécution)
  Dans le projet démo, regardez le fichier `src/main/resources/templates/api/private/user/user.html`. C'est un template *Thymeleaf*.
  Nous pouvons y accéder via http://localhost:8080/api/private/user