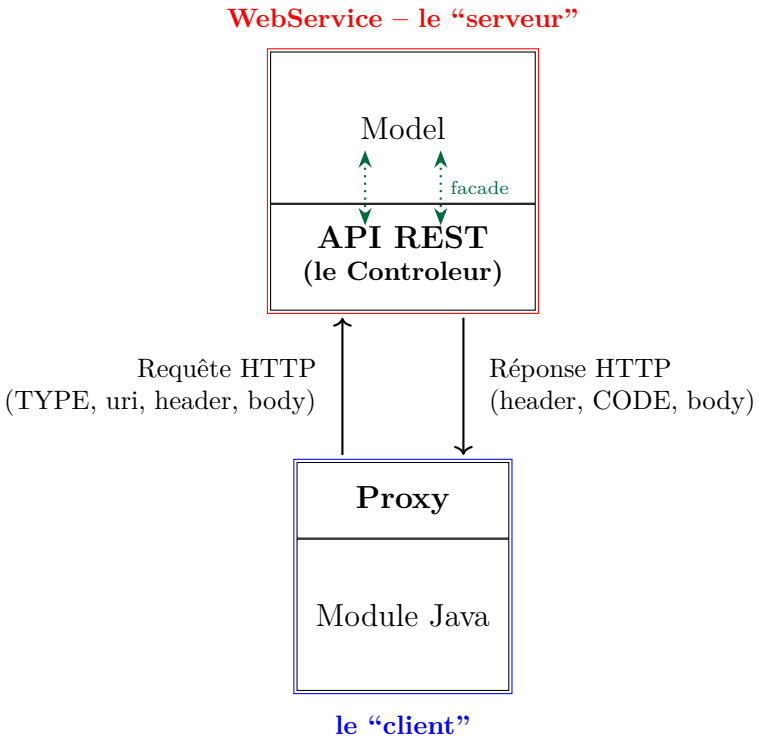
Aide pour les TP WebServices

1 REST en bref

REST est une manière d'écrire une interface de service web, appelée aussi une **API** (Application Programming Interface). Sur une API REST, on envoie des requêtes HTTP, et on reçoit des réponses HTTP. L'API REST est codée dans un controleur, sous la forme de chemins (synonyme de uri, path) associé à un type de requête (GET, PUT, POST, UPDATE, DELETE, etc.). Losqu'une requête est reçue, une methode associée la traite, et retourne une réponse.

ATTENTION : Une API c'est un contrat entre un client et un serveur. Il faut donc respect les types et les paramétres spécifiés par l'API.



Ce qu'il faut retenir :

- Le WebService fait office de serveur.
- Le WebService expose une API, pour recevoir des requêtes HTTPs
- Une requête HTTP c'est : un type, une URI (ou path), un header, un body
- Le WebService répond aux requêtes HTTPs, avec un code et un body
- On envoie des requêtes depuis un client (fichiers HTTPs, code Java, etc.)
- La classe Java en charge d'envoyer des requêtes HTTP s'appelle un proxy

2 Ecrire une requête

2.1 Ce qu'il faut savoir

Une requête c'est un message, qui suit une norme. On appelle cette norme un protocol. En REST, le protocol c'est HTTP.
Une requête HTTP c'est :

- une methode (GET, POST, etc.)
- une URI : un chemin qui décrit l'intention de la requête
- un header : des arguments, des parametres de format d'envoi / de reception
- un corps (un body) : contient les details du message

2.1.1 Les methodes

Les méthodes (que j'ai parfois tendance à appaler les "types" de message HTTP) désignent l'intention. Il en existe plusieurs :

Methode	Equivalence CRUD	Description
GET	READ	Récupère des données sans modification
POST	CREATE	Créer une nouvelle ressource
PUT	UPDATE	Remplace completement une donnée ou une ressource existante
PATCH	UPDATE	Remplace partiellement une donnée ou une ressource existante
DELETE	DELETE	Supprime une ressource

2.1.2 L'URI

Une URI c'est :

- le protocol utilisé
- l'adresse qui heberge le service
- le port sur lequel le service est disponible
- un chemin vers la fonction de l'API REST

Un exemple : `http://localhost:8080/api/user/2928`.
Ici le protocol : `http`, l'adresse : `localhost`, le port : `8080`, et le chemin : `/api/user/2928`.

Note : Des arguments peuvent être passés dans une URI.

2.1.3 Le header

Le header est une suite de clés associées à des valeurs. Classiquement on retrouve :

- `Host` : adresse et port, si il n'est pas défini dans l'URI.
- `Content-Type` : Le format du body de la requête (**Attention** : il peut être imposé dans le controleur)
- `Accept` : le format du body de la réponse
- d'autres variables que vous pouvez définir vous même

2.1.4 Le format dans les messages

Vous pouvez indiquer à le format des corps des requêtes et le format des corps des réponses directement dans le header de votre requête HTTP.

JSON . Le format json est le plus simple. Il ressemble à un dictionnaire en Python. Entre accolade, on fournit une liste de clés et de valeurs. Les valeurs sont typés ! (donc un string : c'est entre guillements, mais un entier : non) On le type application/json. Un exemple

```
{
  "nom": "PHILIPPE",
  "prenom": "Jolan",
  "email": "jolan.philippe1@univ-orleans.fr"
  "age" : 30
}
```

XML . Le format xml est un format structuré, qui utilise des balises. Il est généralement utilisé pour décrire un objet Java. On le type application/xml. Un exemple, pour le type `ProfDto` :

```
<ProfDto>
    <nom>PHILIPPE</nom>
    <prenom>Jolan</prenom>
    <email>jolan.philippe1@univ-orleans.fr</email>
    <age>30</age>
</ProfDto>
```

URL-Encoded . On peut utiliser un format particulier pour passer des arguments directement dans l'URL. Dans ce cas, on peut le passer directement dans l'URI de notre requête, et dans ce cas pas besoin de specifier dans le header de champ `Content-Type`. Pour ajouter les parametres dans l'uri, on utilise ? et on sépare les arguments avec &. Exemple :

```
http://localhost:8080/uri?nom=PHILIPPE&prenom=Jolan&age=30
```

Ou alors, on peut le définir dans le body. Dans ce cas, on le type application/x-www-form-urlencoded. Un exemple :

```
nom=PHILIPPE&prenom=Jolan&age=30
```

Texte On peut aussi juste passer du texte, sans aucune structure, avec le type text/plain

2.1.5 Les réponses

Une réponse HTTP, c'est un code, et un message. Les codes indiquent sous la forme d'un nombre, ce qui s'est passé sur le seveur. Ils sont standardisés. Voilà quelques exemples :

- 200 : OK
- 201 : CREATED
- 400 : BAD REQUEST
- 401 : UNAUTHORIZED
- 404 : NOT FOUND

2.2 en HTTP

En HTTP, tout se déclare simplement en laissant des espaces entre le header, et le body. La réponse du serveur est accessible dans notre client, utilisant `response.headers` pour le header, `response.status` pour le code, et `response.body` pour le corps de la réponse.
Exemple 1 :

```
POST http://localhost:8080/api/cafews/recharge
Content-Type: application/json
Accept: text/plain
cleSecrete: password

{
  "nom": "PHILIPPE",
```

```
  "prenom": "Jolan",
  "email": "jolan.philippe1@univ-orleans.fr"
  "age" : 30
}

> {% client.global.set("myVar", response.body); %}
```

Exemple 2 :

```
GET http://localhost:8080/api/users
Accept: application/json

> {% client.global.set("allUsers", response.body); %}
```

2.3 en Java

Pour écrire une requête Java, on peut utiliser des objets des librairies Spring. On écrit généralement une classe spécifique pour ça : un proxy.

Requête Pour écrire une requête, on utilise la classe `HttpRequest` et sa méthode `newBuilder`. On passe un URI à `newBuilder`, pour obtenir un objet de type `HttpRequest.Builder`.

```
HttpRequest.Builder request = HttpRequest.newBuilder(URI.create("http://localhost:8080/api/
    users"))
```

Cet objet peut être enrichi pour construire l'intégralité de notre requête, étape par étape. On peut par exemple l'enrichir avec une méthode HTTP.

- **Methode** : Par exemple, pour une requête `GET`, on enrichi notre builder avec `.GET()`. On peut evidemment faire pareil avec les autres méthodes HTTP (par ex. `.POST()`, `.PUT()`, etc.).
- **Body** : Cet appel de méthode prend en paramètres le body de la requête.
  (**Note** : la methode `HttpRequest.BodyPublishers.ofString("...")` transforme une chaine de caractères en un body de requête).
- **Header** : La méthode `header()` peut aussi être utilisée pour passer des paramètres/variables au header de la requête.

Une fois la construction de la requête terminé, la méthode `build()` transforme le builder en requête HTTP. Des exemples complets :

```
URI_COMPTES = "http://localhost:8080/api/cafews/compte"
String json = mapper.writeValueAsString(utilisateurDTO);

HttpRequest request1 =
    HttpRequest.newBuilder().uri(URI.create(URI_COMPTES))
                            .header("Content-Type", "application/json")
                            .POST(HttpRequest.BodyPublishers.ofString(json))
                            .build();


HttpRequest request2 =
    HttpRequest.newBuilder().uri(URI.create("http://localhost:8080/api/cafews/compte"))
                            .POST(HttpRequest.BodyPublishers.ofString("nbKilos="+nbKilos))
                            .header("cleSecrete", cleSecrete)
                            .header("Content-Type", "application/x-www-form-urlencoded")
                            .build();
```

Execution et Réponse Lorsqu'un requête est écrite, elle peut être executée et sa réponse traitée comme une chaine de caractères :

```
HttpResponse<String> reponse = client.send(request, HttpResponse.BodyHandlers.ofString());
```

D'un réponse, on peut récuperer le contenu :

- `reponse.statusCode()` retourne le code HTTP (par ex 200) de la réponse
- `reponse.headers()` retourne le header de la réponse
- `reponse.body()` retourne le body de la réponse, sous la forme d'une chaine de caractères.

Note : Si le body est sous une forme JSON, la chaine caractères peut être transformée et manipulée comme une classe Java (si tous les paramètres sont présents) avec `mapper.readValue()` :

```
String json = reponse.body();
UtilisateurDTO utilisateurDTO = mapper.readValue(json, UtilisateurDTO.class)
```

3 Ecrire un controleur

3.1 Modélisation

Avant de se lancer dans le developpement du controleur, qui sera en charge de gérer les requêtes HTTP, on commence par modéliser ce qu'on veut faire. Autrement dit, on fait une liste des fonctionnalités de notre controleur.

3.1.1 Modèle

La première chose à modeliser c'est la donnée qu'on va manipuler dans notre application. Cette donnée est representée sous forme de classe Java. On peut donc modéliser cette partie avec une diagramme UML. Cette partie est relativement simple, et n'est pas vraiment le coeur de ce coeur. C'est du Java, avec de l'orienté objet.

3.1.2 API

Notre WebService doit être capable de gérer un ensemble de requête. L'idée ici est de dresser une liste de requête possible. On peut faire un premier tableau, qui liste pour chaque méthode HTTP, les actions que l'on veut gérer. On associe un nom d'uri, et une méthode, et on donne une description. Le tableau ressemble à ça :

URI	GET	POST	PUT	DELETE
/uri1	Ca fait quoi ?	Ca fait quoi ?	..	..
/uri2	..	..	..	..

Une autre manière de faire, c'est concevoir un tableau plus fin, avec par ligne, une description de point d'accès

Methode	Uri	Input	Response	Description
GET	/uri	au format JSON : nom, prenom, adresse	Un id de type string	Ca fait quoi concretement
POST	/uri2	au format XML : date, idUser	L'ID de l'operation de type int	Ca fait quoi concretement
...				

Ce deuxième tableau ressemble plus à une specification fonctionne de notre API. On est très proche d'une description avec OpenAPI et Swagger. (voir `specification.yaml` dans le TP1, ou `https://editor.swagger.io/`)

3.1.3 DTO

Nos requêtes et réponses vont faire transiter des données structurées. Plutôt qu'utiliser les classes de notre modèle, on utilise des objets particuliers : les DTOs (Data Tansfer Object). Ce sont des classes Java, qui serviront à structurer le contenu des requêtes HTTP.
Imaginons, que l'on veut faire transiter un objet representant un professeur. Alors, on peut écrire une classe `ProfDto` :

```
class ProfDto {
    private String nom;
    private String prenom;
    private String email;
    private int age;
}
```

Le corps XML d'une requête pourra alors ressemble à :

```
<ProfDto>
    <nom>PHILIPPE</nom>
    <prenom>Jolan</prenom>
    <email>jolan.philippe1@univ-orleans.fr</email>
    <age>30</age>
</ProfDto>
```

3.2 Code

Du point de vue WebService, l'API se traduit sous la forme d'un controleur Java. Ce controleur doit se situer dans un package enfant de la classe principale qui démarre votre application.

3.2.1 Mon premier controleur Springboot

Un controleur est une classe Java, où chaque point d'entrée de notre API est associée à une methode Java.

- Il faut donc écrire un controleur, annoté comme controleur avec `@Controleur`.
- En plus de cette annotation, l'annotation `@RequestMapping` donne un préfix à tous les points d'entrée qu'on définira dans la définition de notre controleur.
- Finalement, la facade de notre model (controleur intermediaire en charge de manipuler les classes de notre model) doit être annoté avec `@Autowired`

Un exemple :

```
@RestController
@RequestMapping(value = "/mpws", produces = MediaType.APPLICATION_JSON_VALUE)
public class MonControleur {

    @Autowired
    private FacadeModel facade;

    public MonControleur(FacadeModel facade) {
        this.facade = facade;
    }

    @Autowired
    public void setMonControleur(FacadeModel facade) {
        this.facade = facade;
    }

    ... // Definition de nos fonction pour nos points d'entree

}
```

3.2.2 Mes méthodes et mes points d'entrée

Pour chaque point d'entrée de notre API, on écrit une méthode Java (le nom importe peu) annotée avec :

- le **type de méthode** HTTP (`@GetMapping`, `@PostMapping`, `@PutMapping`, `@DeleteMapping`, ...)
- l'**URI** ciblée (ex : `"/uri/{var}"`).

Les types de paramètres. Si la valeur provient de la requête, on annote les paramètres :

- `@PathVariable("varName")` : variable extraite du chemin (URI)
- `@RequestParam("varName")` : paramètre de requête (dans l'URL après `?` ou dans un `application/x-www-form-urlencoded`)
- `@RequestHeader("varName")` : valeur issue du header HTTP
- `@RequestBody` : corps de la requête (JSON, XML, texte, ...)

Note : si le *nom de la variable Java* est identique à celui du paramètre REST, on peut omettre le nom dans l'annotation (ex : `@PathVariable String id`).
On peut aussi récupérer le *constructeur d'URI* utilisé pour arriver sur ce point d'entrée : `UriComponentsBuilder uriBuilder` (injecté en paramètre de méthode).

```java
// Signatures minimales par type
@GetMapping(value="/uri/{var}")
public ResponseEntity<?> getQuelqueChose(@PathVariable("var") String var) {}

@PostMapping(value="/uri/{var}")
public ResponseEntity<?> postQuelqueChose(@PathVariable("var") String var) {}

@PutMapping(value="/uri/{var}")
public ResponseEntity<?> putQuelqueChose(@PathVariable("var") String var) {}

@DeleteMapping(value="/uri/{var}")
public ResponseEntity<?> deleteQuelqueChose(@PathVariable("var") String var) {}
```

Exemple complet (paramètres taggés, sans corps de méthode).

3.2.3 Format d'entrée et de sortie

Les annotations de méthode peuvent préciser le `Content-Type` *accepté* côté entrée (`consumes`) et les formats *retournés* (`produces`).
Quelques cas fréquents :

- `consumes = "application/json"` pour un **body JSON** (`@RequestBody`).
- `consumes = "application/x-www-form-urlencoded"` pour un body type formulaire.
- `produces = "application/json"` si l'API **renvoie du JSON** par défaut.

Exemple GET qui produit du JSON ou du XML selon `Accept` :

4 Gestion d'erreur

Bien lire le message de sortie ! Le code erreur indique la cause la plus probable. Erreurs communes :

- **Mauvais types** (ex : chaîne fournie au lieu d'un entier)
  -> vérifier le DTO, les convertisseurs, et le `Content-Type`.
- **Mauvaise URI** (typo, préfixe `@RequestMapping` manquant ou incorrect)
  -> vérifier les valeurs `@RequestMapping` et `@GetMapping/...`.
- **Paramètres mal placés** (ex : valeur envoyée en header alors qu'attendue en body)
  -> vérifier `@RequestParam` vs `@RequestHeader` vs `@RequestBody`.
- **Content-Type / Accept incompatibles**
  -> vérifier `consumes`/`produces` sur la méthode et les en-têtes envoyés.
- **404 NOT FOUND** : mauvaise route ou `@RequestMapping` non chargé (package non scanné).
- **415 UNSUPPORTED MEDIA TYPE** : `Content-Type` ne correspond pas à `consumes`.
- **406 NOT ACCEPTABLE** : aucun `produces` compatible avec l'en-tête `Accept`.
- **400 BAD REQUEST** : payload invalide / champs manquants / JSON mal formé.
- **500 INTERNAL SERVER ERROR** : exception côté serveur (NullPointerException, etc.).

Astuce : active les logs de requêtes/réponses et teste avec des fichiers HTTP (ou cURL/Postman) pour isoler rapidement le trio *URI + headers + body*.

```java
@GetMapping(value="/users/{id}")
public ResponseEntity<UserDto> getUser(
        @PathVariable("id") String id,
        @RequestParam(name = "verbose", required = false, defaultValue = "false") boolean
            verbose,
        @RequestHeader(name = "X-Trace-Id", required = false) String traceId,
        UriComponentsBuilder uriBuilder
) {}

// Meme exemple en s'appuyant sur l'egalite des noms (id == {id})
@GetMapping(value="/users/{id}")
public ResponseEntity<UserDto> getUser2(
        @PathVariable String id, // nom identique -> parametre facultatif dans l'annotation
        @RequestParam(required = false, defaultValue = "false") boolean verbose,
        @RequestHeader(name = "X-Trace-Id", required = false) String traceId,
        UriComponentsBuilder uriBuilder
) {}

// POST avec body JSON
@PostMapping(value="/users")
public ResponseEntity<UserDto> createUser(
        @RequestBody CreateUserDto dto,
        UriComponentsBuilder uriBuilder
) {}

// PUT avec PathVariable + body
@PutMapping(value="/users/{id}")
public ResponseEntity<UserDto> updateUser(
        @PathVariable String id,
        @RequestBody UpdateUserDto dto
) {}

// DELETE avec parametre dans la query string
@DeleteMapping(value="/users")
public ResponseEntity<Void> deleteUser(
        @RequestParam("email") String email
) {}
```

```java
@PostMapping(
    value = "/prof",
    consumes = { "application/xml" },
    produces = { "application/json", "application/xml" }
)
public ResponseEntity<ProfDto> createProf(
        @RequestBody ProfDto prof
) {}
```

```java
@GetMapping(
    value = "/prof/{id}",
    produces = { "application/json", "application/xml" }
)
public ResponseEntity<ProfDto> getProf(@PathVariable String id) {}
```