WebServices REST

WebServices

Un WebService (qu'on notera WS à partir de maintenant) est un programme qui tourne sur un serveur web. Il renvoie des données à un client qui l'interroge.

On voit qu'un site web est une variante de WS : le client est un browser et le site web renvoie aussi des données (en HTML).

Plus spécifiquement, un WS renvoie des données sous une certaine forme quand le client fait ses requêtes sous une certaine forme.

On voit donc qu'il existe plusieurs formes de WS selon la façon dont le client appelle et la façon dont le client répond. Plus exactement il y a deux choses qu'il faut distinguer :

- Le protocole d'échange : sous quelle forme transitent les *ordres* entre client et WS
- Le format des données : sous quelle forme transitent les *données* entre client et WS

En gros il existe deux sortes de WS :

- Les WS SOAP : le protocole d'échange entre client et WS est du SOAP et les données transitent en XML
- Les WS REST : le protocole d'échange est du REST et les données transitent en JSON.

Actuellement les WS REST ont le vent en poupe et on va s'occuper d'eux.

Le protocole REST (ou API REST)

Pour bien saisir l'API REST, prenons le cas canonique d'un WS qui fait un CRUD. Et puisqu'on l'a déjà utilisé, on va prendre le cas de Specie.

L'API REST est la suivante :

- GET /rest/specie : renvoie la liste des Specie
- GET /rest/specie/2 : renvoie la Specie d'id 2
- DELETE /rest/specie/4 : détruit la Specie d'id 4
- POST /rest/specie : crée une Specie
- PUT /rest/specie/6 : update la Specie d'id 6

On voit 2 choses :

- D'abord, il existe deux commandes supplémentaires qui n'apparaissaient pas quand on s'occupait de sites web : PUT et DELETE (il en existe d'autres, mais ce n'est pas le sujet). On voit à quel point l'API REST est « parlante » : le nom de la commande indique simplement ce que doit faire le WS.
- On voit que les url ne référencent pas directement la racine du site (on ne fait pas PUT /6 par exemple). Il est de bon ton que l'url soit composé de deux parties :
    - Un préfixe qui indique (par exemple) sur quelle partie du site on travaille. Après tout sur un même site, on peut avoir à la fois un site web, un ou des WS REST et un ou des WS SOAP. Ici on a juste mis /rest pour signifier qu'on va utiliser la partie REST du site.
    - Un suffixe qui indique sur quoi on travaille. Ici on a mis specie. Si le WS permettait aussi de travailler sur Animal, on aurait par exemple, GET /rest/animal qui renvoie la liste des Animal.

ATTENTION : la différentiation des url en plusieurs parties n'est PAS une obligation. C'est juste une bonne pratique. Et la façon d'écrire les url peut varier d'un projet à l'autre.

JSON

On a déjà vu le JSON, mais on va refaire un petit tour.

Soit une Class Specie

```
public class Specie {
    Long id;
    String commonName;
    String latinName;

…….

}
```

Si on fait :

```
Specie sp = new Specie (12L, "common","latin") ;
```

Alors l'objet JSON associé sera :

```
{
        "id": 12,
        "commonName": "common",
        "latinName": "latin"
}
```

On voit donc qu'un objet JSON est de la forme

```
{
        "nom_du_champ_1": valeur_du_champ1,
        "nom_du_champ_2": valeur_du_champ2,
        …
        "nom_du_dernier_champ": valeur_du_dernier_champ
}
```

Remarquez :

- Le *nom* des champs est toujours entre "" mais la *valeur* n'est entre "" que si c'est une String.
- Les lignes de JSON finissent par , mais pas la dernière.

- Les blancs n'ont pas d'importance

Soit la Class Animal :

```
public class Animal {
    Long id;
    String name;
    String color;
    Specie specie;

…….

}
```

Si on fait :

```
Animal an = new Animal(1L, "myName","myColor",new Specie (12L, "common","latin")) ;
```

Alors l'objet JSON associé sera :

```
{
        "id": 1,
        "name": "myName",
        "color": "myColor",
        "specie": {"id":12, "commonName":"common","latinName":"latin"}
}
```

On voit donc qu'on peut imbriquer des objets dans des objets.

On peut aussi avoir une liste (ou un array) d'objets (ici une liste de Specie) :

```
[

    {
        "id": 1,
        "commonName": "Chat",
        "latinName": "Felis silvestris catus"
    },

    {
        "id": 2,
        "commonName": "Chien",
        "latinName": "Canis lupus familiaris"
    },

……
]
```

On voit que les listes sont délimitées par des []. Evidemment, on peut avoir des listes dans des objets, des listes d'objets (comme c'est le cas ici), etc.

Le Controller

Le Controller pour un WS est très similaire à celui d'un site web. On ne sera donc pas dépaysé.

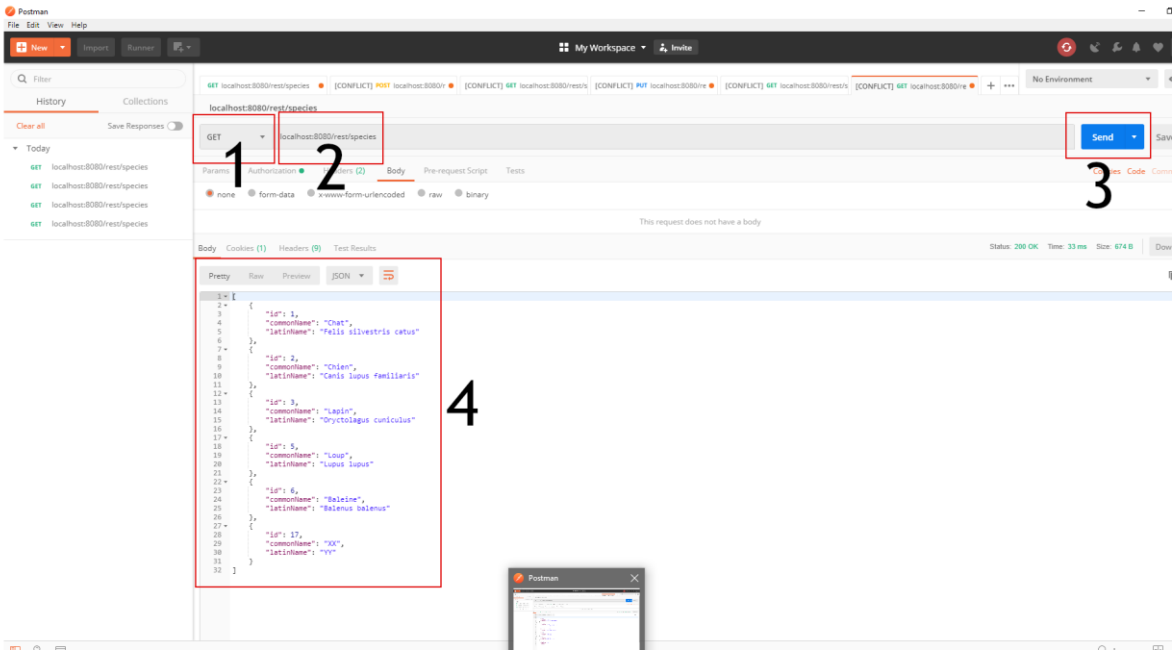
```
On indique qu'il s'agit d'un Controller REST
@RestController
On indique la base de l'url pour le Controller. Cela signifie que toutes les url commençant par
/rest/species seront prises en compte par ce Controller.
@RequestMapping("/rest/species")
class SpecieController {
    ……..
    Comme l'URL REST est /rest/species pour récupérer la liste des Specie, le Mapping associé est vide
(puisque la base URL est déjà /rest/specie).
    @GetMapping("")
    public List<Specie> list() {
        List<Specie> list = service.list();
        return list;
```

```
    }
    Dans le cas du POST ou du PUT, on récupère des données (latinName et CommonName en pratique). Il faut dire où sont ces données dans la requête pour que le JSON puisse être transformé en un objet Specie. Ici on dit que les données sont dans le corps de la Requête (c'est le cas classique en JSON) ; On indique donc cela avec @RequestBody.
    @PostMapping("")
    public long create(@RequestBody Specie sp) {
        service.save(sp);
        C'est pas mal de renvoyer au client l'id du Specie nouvellement créé.
        return sp.getId();
    }
    Cas du DELETE
    @DeleteMapping("/{id}")
    public void delete(@PathVariable("id") long id) {
        service.deleteById(id);
    }
    ……..
}
```

Tester

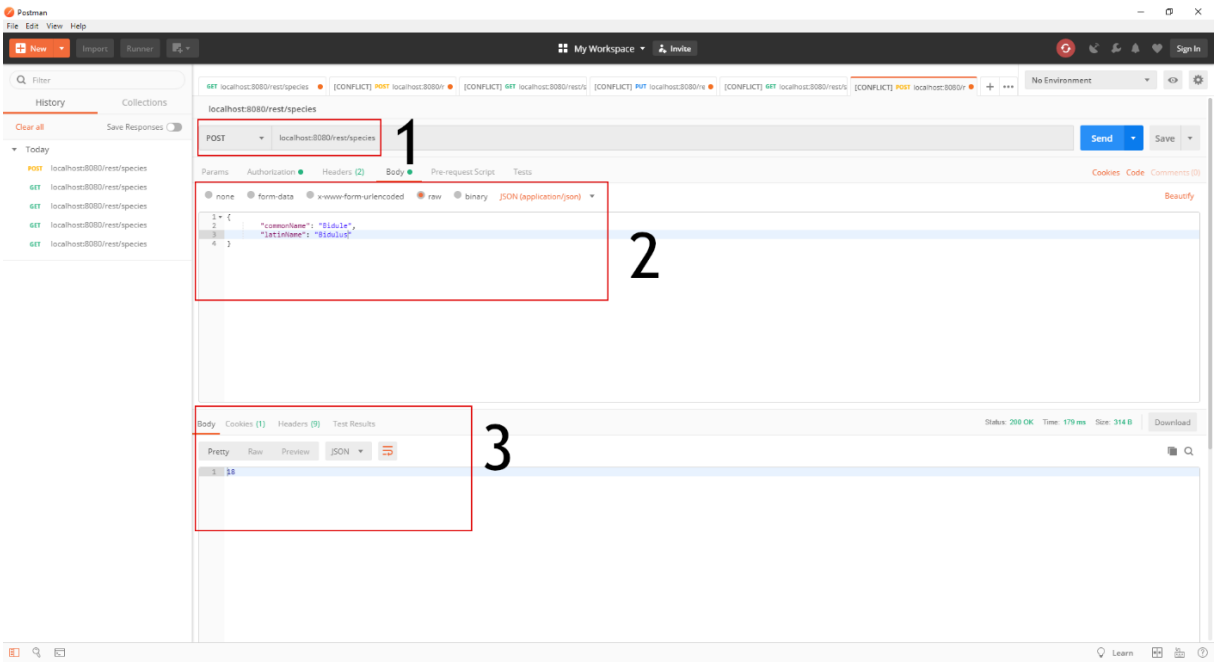
Comme on l'a vu, il faut que quelque part il y ait un client REST de façon à récupérer les données JSON issues du serveur. Typiquement le client est un bout de code dans un programme JAVA ou bien un bout de code JS (dans Angular par exemple).
Ceci étant, pour tester le serveur, il est bon d'utiliser un programme dédié aux tests.

A titre d'exemple, on va utiliser Postman, mais d'autres programmes sont possibles.
Ici un exemple pour avoir la liste des Specie.



On choisit le type de requête, puis on tape l'url de la requête, et enfin, on appuie sur *Send*. La requête est envoyée au serveur et on voit la réponse en 4.

Dans le cas d'un POST (ou d'un PUT), il faut que des données remontent du client. Comme on a vu que ces données sont dans le corps de la requête (@RequestBody au niveau du Controlleur), c'est là qu'il faut remplir les données.



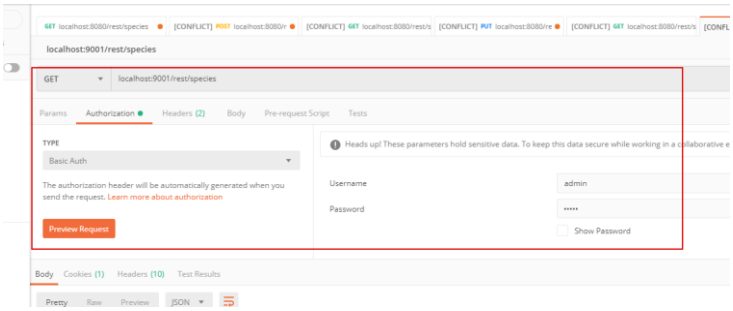
Imaginons qu'on choisisse de faire un POST et de créer une nouvelle Specie :

- En 1, on choisit donc la bonne requête
- En 2, On sélectionne *Body*, puis *Raw* pour le *Body* (pas de codage particulier pour les données du *Body*), on remplit donc ce *Body* avec les données pour créer une nouvelle Specie.
- En 3, on a le retour du WS (ici l'id de la Specie nouvellement créée).

Sécurité

Comme on le voit, on n'a ici aucune sécurité : n'importe qui peut se connecter, récupérer les données, créer des Specie, les modifier, etc …
C'est très mauvais du point de vue sécurité.
Dans un site Web classique, on devrait passer par une page d'identification pour avoir des droits sur le site. Comment fait-on avec un WS ?

L'idée est de passer les identifiants (login + mot de passe) de l'utilisateur dans le Header de la requête HTTP.
Pour faire simple, on va utiliser l'Authentification Basic qui envoie le login+Mdp sous forme login:mdp codé en Base64 dans le Header, ce qui fait qu'on se retrouve avec quelque chose du genre `Authorization: Basic QWxhZGRpbjpPcGVuU2VzYW1l` dans le Header.



Dans Postman, on sélectionne l'onglet *Authorization* et on remplit les champs *Username* et *Password*.

Et au niveau du WS, comment va-t-on gérer cette autorisation ? Eh bien, comme Spring Boot vous facilite le travail, il suffit de créer deux classes.

Mais avant toute chose, il faut ajouter une dépendance dans le pom.xml

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

Un truc à remarquer : il suffit de rajouter cette dépendance pour le WS soit sécurisé, c.a.d demande un login et un mot de passe à son client.

```
Ici on crée la classe qui gère la configuration du WS. Elle dérive donc de
WebSecurityConfigurerAdapter.
@Configuration
On dit que la securité est active
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

	On déclare un AuthenticationEntryPoint
	@Autowired
	private AuthenticationEntryPoint authEntryPoint;
	A partir de Spring Boot 2.0, les login/Mdp sont sur-encryptés via le protocole BCrypt. Il faut
donc déclarer un système de cryptage BCrypt.
	@Autowired
	BCrypt crypt;
	Là, on dit que les requêtes nécessitent une authentification en Basic et, en cas d'erreur
envoie sur le authenticationEntryPoint.
	@Override
	protected void configure(HttpSecurity http) throws Exception {
		http.csrf().disable().authorizeRequests()
		.anyRequest().authenticated()
		.and().httpBasic()
		.authenticationEntryPoint(authEntryPoint);
	}
	Là, on définit les user/mdp acceptés (user/user et admin/admin). On leur donne aussi un role,
ce qui n'a pas d'utilité pour l'authentification (mais qui en aura pour les authorisations).
	@Override
	protected void configure(AuthenticationManagerBuilder auth) throws Exception {

	auth.inMemoryAuthentication().withUser("user").password(crypt.encode("user")).roles("USER").and
().
				withUser("admin").password(crypt.encode("admin")).roles("ADMIN");

}
```

Pour le BCrypt, la solution la plus évidente est de créer une classe dérivée de `BCryptPasswordEncoder` et de l'annoter avec @Component. Une solution plus élégante est d'utiliser une fonctionnalité de Spring que nous n'avons pas encore vue :

```
@Bean
public PasswordEncoder  encoder() {
    return new BCryptPasswordEncoder();
}
```

Le @Bean signifie que l'objet crée par la fonction va être un Bean, soit une sorte de @Component. Dans ces conditions l'objet crypt est inutile et dans la fonction *configure()*, au lieu de *crypt.encode(xxx)*, on aura *encoder().encode(xxx)*.

La deuxième classe indique ce qu'il se passe en cas d'erreur d'authentification.

```
@Component
La classe en question implémente donc AuthenticationEntryPoint qu'on a vu dans la classe d'avant.
public final class RestAuthenticationEntryPoint
  implements AuthenticationEntryPoint {

    @Override
    public void commence(
        final HttpServletRequest request,
        final HttpServletResponse response,
        final AuthenticationException authException) throws IOException {
        Et en pratique, en cas d'erreur on renvoie le code HTTP 401 (on peut aussi renvoyer un 403)
        response.sendError(HttpServletResponse.SC_UNAUTHORIZED,
          "Unauthorized");
    }
}
```

Securité SQL

Dans l'exemple précédent, on voit que les login/mdp autorisés sont écrits en dur dans le code, ce qui est assez moyen. On pourrait certes récupérer les login/mdp dans un fichier de properties (par exemple) et coder comme vu dans l'exemple précédent, mais ça reste pas terrible.

Comme Spring Boot vous aime, il vous permet très facilement d'utiliser des login/mdp stockés dans une base SQL (ou dans un annuaire LDAP).

Comme en général, on utilise déjà une BDD (pour récupérer les données utilisées par le WS), il suffit de rajouter deux tables dans la BDD :

- users qui contient effectivement les login/mdp. Ses champs sont
  - username
  - password (codé en BCrypt)
  - enabled
- authorities qui définit les rôles
  - username (voir plus haut)
  - authority (le role)

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

        @Autowired
        private AuthenticationEntryPoint authEntryPoint;
```

```
        Ici, on déclare la Datasource, c.a.d la liaison vers la BDD.
        @Autowired
        DataSource dataSource;

        ……
        Ici on dit qu'on utilise un système d'authentification jdbc via la dataSource.
        @Override
        protected void configure(AuthenticationManagerBuilder auth) throws Exception {
                auth.jdbcAuthentication().passwordEncoder(encoder()).dataSource(dataSource);
        }

……

}
```

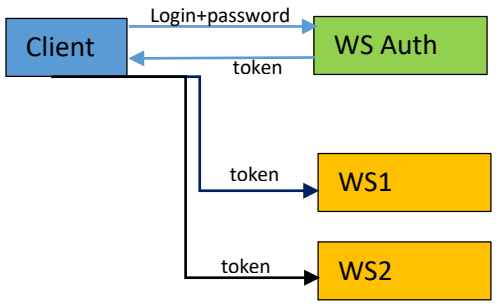
JWT

(Pour plus de détails, voir par exemple : https://jwt.io/introduction/)

JWT (JSON web Token) est un système d'identification à token. Qu'est-ce que cela signifie ?

Imaginons qu'on ait 3 WS. Si on veut s'identifier auprès de ces 3 WS en utilisant les méthodes précédentes, il faut donc que chaque WS ait une BDD d'identification, et donc que chacun ait les mêmes tables dupliquées (avec mise à jour si des utilisateurs sont ajoutés/modifiés). Ce n'est pas très pratique et une BDD commune aux 3 WS n'est pas toujours possible à faire.

L'idée est donc la suivante : On va se connecter à un WS particulier en plus, le WS d'identification qui va renvoyer un truc, que l'on appelle un token, qui va servir pour s'authentifier auprès des 3 autres.



En pratique le token est une chaine codée en Base64 qu'on renvoie dans le Header (en mode Bearer) de la Requête REST vers les WS.

Le token est composé de 3 parties :

- Le HEADER : Il contient essentiellement le type d'encodage pour la SIGNATURE (SHA256 ou RSA)
- Les DONNEES (Payload) : Typiquement
    - Le nom du user connecté
    - Ses rôles
    - La date d'expiration du token
- La SIGNATURE : Quelque chose de crypté qui permet de vérifier que le token est bien un token autorisé et pas un truc employé par un hacker. En pratique, elle est générée à partir d'une phrase clé (secret en anglais) que les 3 WS connaissent. C'est quand même plus pratique que d'avoir 3 fois la même BDD.

EXERCICE : Pour se mettre dans une situation réelle d'entreprise, vous allez chercher sur internet des exemples permettant d'implémenter un système JWT sur le REST de Specie. Les contraintes sont les suivantes :

- Ne pas utiliser un exemple qui emploie en plus OAuth (c'est VRAIMENT compliqué dans ce cas là)
- Normalement les exemples implémentent un système avec une BDD. Ce qui est la bonne chose à faire. Pour voir si vous comprenez ce que vous faites (et pas juste du Copier/Coller), ne pas utiliser une BDD et faire en sorte que le système d'authentification utilise 2 utilisateurs en dur (disons userjwt/userjwt et adminjwt/adminjwt).