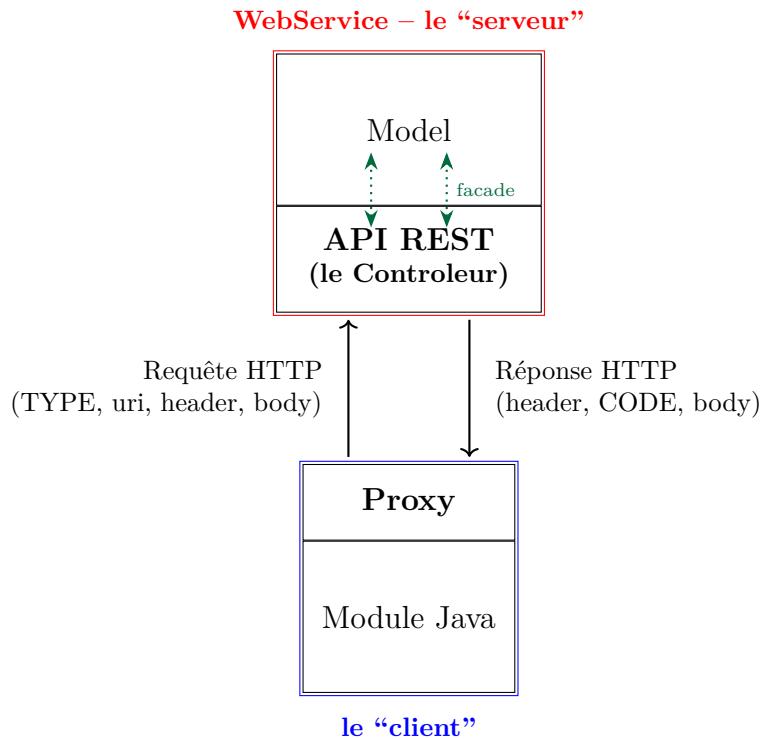


Aide pour les TP WebServices

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 contrôleur, sous la forme de chemins (synonyme de uri, path) associé à un type de requête (GET, PUT, POST, UPDATE, DELETE, etc.). Lorsqu'une requête est reçue, une méthode associée la traite, et retourne une réponse.

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



Ce qu'il faut retenir :

- Le WebService fait office de serveur.
- Le WebService expose une API, pour recevoir des requêtes HTTP.
- Une requête HTTP c'est : un type, une URI (ou path), un header, un body.
- Le WebService répond aux requêtes HTTP, avec un code et un body.
- On envoie des requêtes depuis un client (fichiers HTTP, 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 méthode (GET, POST, etc.)
- une URI : un chemin qui décrit l'intention de la requête
- un header : des arguments, des paramètres de format d'envoi / de réception
- un corps (un body) : contient les détails du message

2.1.1 Les méthodes

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

Méthode	Équivalence CRUD	Description
GET	READ	Récupère des données sans modification
POST	CREATE	Créer une nouvelle ressource
PUT	UPDATE	Remplace complètement 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 protocole utilisé
- l'adresse qui héberge 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 protocole : `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 contrôleur)
- `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 la fois 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 spécifier dans le header de champ `Content-Type`. Pour ajouter les paramètres 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 serveur. 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 évidemment 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 méthode `HttpRequest.BodyPublishers.ofString("...")` transforme une chaîne 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 chaîne de caractères :

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

D'un réponse, on peut récupérer 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 chaîne de caractères.

Note : Si le body est sous une forme JSON, la chaîne 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 contrôleur

3.1 Modélisation

Avant de se lancer dans le développement du contrôleur, 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 contrôleur.

3.1.1 Modèle

La première chose à modéliser c'est la donnée qu'on va manipuler dans notre application. Cette donnée est représentée sous forme de classe Java. On peut donc modéliser cette partie avec un diagramme UML. Cette partie est relativement simple, et n'est pas vraiment le cœur de ce cœur. 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 concrètement
POST	/uri2	au format XML : date, idUser	L'ID de l'opération de type int	Ca fait quoi concrètement
...				

Ce deuxième tableau ressemble plus à une spécification fonctionnelle 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 Transfer Object). Ce sont des classes Java, qui serviront à structurer le contenu des requêtes HTTP.

Imaginons, que l'on veut faire transiter un objet représentant 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 ressembler 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 contrôleur Java. Ce contrôleur doit se situer dans un package enfant de la classe principale qui démarre votre application.

3.2.1 Mon premier contrôleur Springboot

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

- Il faut donc écrire un contrôleur, annoté comme contrôleur avec `@Controller`.
- 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 contrôleur.
- Finalement, la façade de notre modèle (contrôleur intermédiaire en charge de manipuler les classes de notre modèle) 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;
    }

    ... // Définition de nos fonction pour nos points d'entrée
}
```

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).

```
// 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*.

```

@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
) {}

```

5 Sécurité avec authentification (sans JWT/JWS)

Ici, on reste dans une approche **stateless** : le serveur ne stocke pas de session.

Attention : sans HTTPS, HTTP Basic expose (en pratique) vos identifiants. En TP c'est OK, en prod : **HTTPS obligatoire**.

5.1 Écrire une requête authentifiée

5.1.1 En HTTP

Le principe : envoyer un header `Authorization: Basic args`

```

GET http://localhost:8080/mpws/prof/12
Accept: application/json
Authorization: Basic jolan.philippe1@univ-orleans.fr miage2025

{
    ... # body
}

```

```

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

```

```

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

```

5.1.2 En Java (proxy)

On ajoute le header `Authorization`.

```

String username = "...";
String password = "...";
String authHeader = "Basic " + username + " " + password ;
HttpRequest request =
    HttpRequest.newBuilder()
        .uri(URI.create("http://localhost:8080/mpws/prof/12"))
        .header("Accept", "application/json")
        .header("Authorization", authHeader)
        .GET().build();

```

5.2 Les configurations

5.3 Les rôles

5.3.1 Mapper un utilisateur à des rôles

```

@Service
public class CustomUserDetailsService implements UserDetailsService {

    ... // facade du model

    @Override
    public UserDetails loadUserByUsername(String username)
        throws UsernameNotFoundException {

        Utilisateur utilisateur = ... ; // recuperer depuis le model
        if (utilisateur == null) throw new UsernameNotFoundException("...");

        String[] roles = ... ; // via une methode, ou le model directement

        return User.builder()
            .username(utilisateur.getEmail())
            .password(utilisateur.getEncodedPassword())
            .roles(roles[0], roles[1], ...)
    }
}

```

```

        .build();
    }
}
```

5.3.2 Donner des permissions en fonction des rôles

```

@Configuration
public class SecurityConfig {

    @Bean
    protected SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http.csrf(CsrfConfigurer::disable) // utile en REST stateless
            .authorizeHttpRequests(authorize -> authorize
                // Exemples de règles de routes:
                // .requestMatchers(HttpMethod.METHOD, uri).permitAll()
                // .requestMatchers("uri/**").hasRole(ROLE1)
                // .requestMatchers("uri/**").hasAnyRole(ROLE1, ROLE2)
                // .requestMatchers("uri/**").authenticated()
                .anyRequest().denyAll()
            )
            .httpBasic(Customizer.withDefaults()) // active Basic Auth
            .sessionManagement(session ->
                session.sessionCreationPolicy(SessionCreationPolicy.STATELESS)
            );
        return http.build();
    }
}
```

Remarques :

- `permitAll()` : accessible sans login.
- `authenticated()` : login requis, peu importe le rôle.
- `hasRole("X")` : login requis + rôle `ROLE_X` (Spring ajoute souvent le préfixe `ROLE_`).
- `denyAll()` : pratique pour éviter d'exposer accidentellement une route oubliée.

5.4 Encoder (hasher) des mots de passe

On ne stocke jamais un mot de passe en clair. On stocke une version encodée. Spring fournit `PasswordEncoder`. Le plus classique : `BCrypt`.

```

@Configuration
public class CryptoConfig {

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

Usage typique (idée) :

- à l'inscription : `encoded = encoder.encode(mdpClair)` puis stockage dans la facade.
- à l'authentification : Spring compare `encoder.encode(mdpClair)` vs `encoded` automatiquement via le `PasswordEncoder`.

6 Sécurité un Token (authentification cryptée) en Spring Security 6

6.1 Écrire une requête authentifiée

Le principe : Obtenir un *token* à la connexion, puis le renvoyer à chaque requête sur les routes protégées. Le client conserve le token et le place ensuite dans un header HTTP (`Authorization`). Le serveur ne mémorise toujours pas de session : tout est dans le token (au minimum l'identité, souvent des rôles), dont il vérifie la signature à chaque requête.

Exemple :

```
POST localhost:8080/api/login
Content-Type: application/json

{
    "email": "jolan.philippe1@univ-orleans.fr",
    "password": "darkvador"
}

> {% client.global.set("token", response.headers.valueOf("Authorization")); %}

###

GET localhost:8080/api/utilisateurs
Accept: application/xml
Authorization: {{token}}
```

Note : le principe est le même en Java, voir les requêtes précédentes.

6.2 Configurer son Web Service

6.2.1 Spécifier le protocol d'identification (avec oauth2)

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http.csrf(CsrfConfigurer::disable)
        ...
        .oauth2ResourceServer((oauth2) -> oauth2.jwt(Customizer.withDefaults()))
    return http.build();
}
```

6.2.2 Encoder et décoder le token

Créer la clé cryptographique avec une méthode `getJWK` utilisée pour **signer** les tokens et/ou pour **vérifier** leur signature. Le type de clé doit être cohérent avec l'algorithme choisi (par ex. HMAC pour HS256, ou paire de clés pour RS256/ES256). En pratique, la clé doit être stable (ne pas changer à chaque redémarrage) sinon tous les tokens précédents deviennent invalides. Dans `CryptoConfig`, on peut ajouter la méthode :

```
public JWK getJWK() {...}
```

Depuis cette clé, écrire une méthode de serialization et de déserialization du token. Ca fait aussi d'encodage, et de décodage :

- `JwtEncoder` : prend des claims + un header JWS, signe et produit une chaîne JWT (un token encodé).
- `JwtDecoder` : prend une chaîne JWT, vérifie la signature, puis reconstruit un objet `Jwt` (claims accessibles), et déclenche une erreur si le token est invalide/expiré.

```

@Bean
public JwtEncoder jwtEncoder(JWK jwk) {...}

@Bean
public JwtDecoder jwtDecoder(JWK jwk) {...}

```

6.2.3 Générer un token à la connexion

On veut centraliser la **construction des claims** et produire le token renvoyé au client après authentification réussie. Cette méthode se construit à partir de :

- **issuer** : qui émet le token (identifie le serveur émetteur)
- **issuedAt** : date d'émission
- **expiresAt** : date d'expiration
- **subject** : l'identité principale (souvent email ou username)
- **claim(...)** : des infos supplémentaires (ex : rôles, identifiant interne)

Enfin, elle appelle `jwtEncoder(...)` pour **signer** le tout et retourner le `tokenValue` (la chaîne envoyée au client). Un exemple :

```

@Bean
Function<Utilisateur,String> genereTokenFunction(JWK jwk) {

    return personne -> {

        Instant now = Instant.now();
        long expiry = 36000L;

        /*
        Dans les 4 lignes de code suivantes, on:
        - Recupere les roles
        - On cree un objet stream pour utiliser la fonction map
        - On utilise map (applique la meme fonction a tous les elements du stream)
        - On recupere tous les elements du stream en une seule chaine de caracteres
        */
        String[] roles = getRoles(personne.getEmail());
        Stream<String> stream_roles = Arrays.stream(roles);
        Stream<String> stream_roles_afterToString = stream_roles.map(x -> x.toString());
        String scope = stream_roles_afterToString.collect(Collectors.joining(" "));

        JwtClaimsSet claims = JwtClaimsSet.builder()
            .issuer("self")
            .issuedAt(now)
            .expiresAt(now.plusSeconds(expiry))
            .subject(personne.getEmail())
            .claim("scope", scope)
            .claim("idUtilisateur",String.valueOf(personne.getIdUtilisateur()))
            .build();

        JWSHeader.Builder b = new JWSHeader.Builder(JWSAlgorithm.ES256);
        JwsHeader myJwsHeader = JwsHeader.with(MacAlgorithm.HS256).build();
        return jwtEncoder(jwk).encode(JwtEncoderParameters.from(myJwsHeader, claims)).
            getTokenValue();
    };
}

```

Finalement, on défini une méthode `jwtAuthenticationConverter` pour transformer le Jwt décodé en un Authentication Spring Security. Concrètement, il lit des claims (souvent `scope` ou `roles`) et les convertit en `GrantedAuthority` pour que `hasRole(...)` fonctionne correctement.

```
@Bean
public JwtAuthenticationConverter jwtAuthenticationConverter() {
    JwtGrantedAuthoritiesConverter grantedAuthoritiesConverter = new
        JwtGrantedAuthoritiesConverter();
    grantedAuthoritiesConverter.setAuthorityPrefix("ROLE_");

    JwtAuthenticationConverter jwtAuthenticationConverter = new JwtAuthenticationConverter()
        ;
    jwtAuthenticationConverter.setJwtGrantedAuthoritiesConverter(grantedAuthoritiesConverter
        );
    return jwtAuthenticationConverter;
}
```