Sécuriser une API REST

Plan

- Sécuriser une API REST
- Utilisation de Spring Security

Sécuriser une API REST

Sécuriser une API REST

Coeur de l’application

Les Applications front end sont généralement le point d’entrée des utilisateurs vers un système d’informations (SI).

Les **APIs** forment le plus souvent le **coeur du système**, au centre d’un ensemble d’applications front end.

Elles portent la cohérence des données, et peuvent exposer des informations sensibles (données client, informations médicales, …)

Exposition des données

N’importe qui ne doit pas pouvoir y accéder !

Cette accès restreint peut se faire de deux manières, souvent utilisées en conjonction :

- Par l’infrastructure
- Par du contrôle d’accès

API Key

Contrôle d’accès

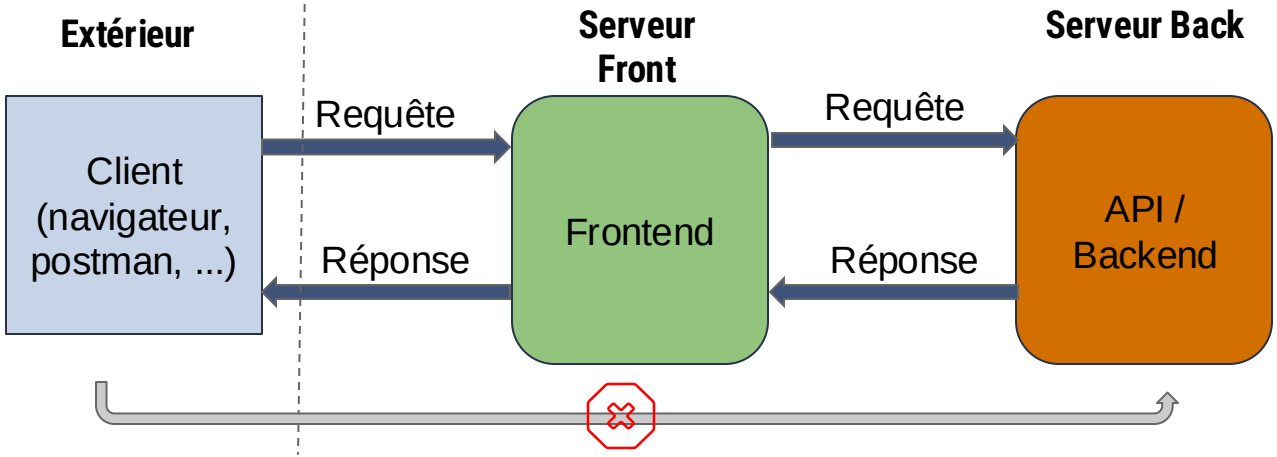
Parfois il est nécessaire de filtrer assez finement les actions:

Certains utilisateurs peuvent avoir besoin de créer des données, d’autres de le modifier, etc.

D’autres encore ne pourraient que consulter, en lecture seule, l’information.

Cet affinement des droits est appelé : **Contrôle d’accès**

Par l'infrastructure



Par l’infrastructure

Filtrer les requêtes entrantes

Cette approche consiste à n’autoriser que certaines adresses à se connecter au serveur “Back” hébergeant l’API.

On parle ici d’une **plage d’IP filtrée ou autorisée**. Ce modèle définit une liste de machines de confiance, dont les front end applicatifs.

Dans cette approche, seul un passage par le front end, faisant “proxy”, permet d’appeler l’API.

Exposition des données

Une sécurité insuffisante

Les appels au serveur frontend sont faits en Javascript depuis le poste client.

Il est donc techniquement possible de **modifier les appels faits directement depuis le client** ! Et donc de récupérer ou modifier des données en théorie inaccessible !

Exposition des données

Une sécurité insuffisante

Exemple :

Mon application récupère mes informations clientes via l'appel :

- http://monappli.com/proxy/client/**228**

En changeant un peu de Javascript, ou en utilisant directement l'adresse ci dessous, je peux récupérer des données clientes qui ne sont pas les miennes :

- http://monappli.com/proxy/client/**127**

Exposition des données

Une sécurité insuffisante

Cette solution permet de se protéger des attaques externes d'utilisateurs n'ayant pas accès à l'application front end.

Pour bien faire, il faudrait que l'API vérifie également que l'utilisateur a les droits d'appeler une ressource spécifique.

Par l’application

Comment s’authentifier sur une API ?

Dans une application Web, la méthode d’authentification la plus courante est le formulaire login/password.

Pour une API Rest, cette approche est impossible car aucune page HTML ne peut être utilisée.

Il faut donc envoyer à l’API les identifiants autrement. Plusieurs solutions existent, chacune d’entre elle donnant naissance à une authentification différente.

Par l’application

Méthodes d’authentification :

- Utilisation d’une API Key
- Authentification “Basic”
- Utilisation d’un Token
- Utilisation d’un serveurs tiers d’authentification

1/ API Key

La méthode la plus simple

Une **API Key** (clé d'API) est une information envoyée par un client pour dire qui il est.

Cette information est en générale ajoutée aux headers de la requête : **x-api-key** par exemple. Parfois, aux paramètres de la requête : http://test.com**?apikey=**

La clé est généralement fournie par le responsable de l'API à ses clients.

1/ API Key

Identification, pas authentification

En règle général, cette information sert à identifier un client plus qu'à l'authentifier. Typiquement une clé d'API sert à :

- **Éviter le trafic anonyme**, venu de l'extérieur si votre API n'est pas sécurisée par l'architecture
- **Contrôler le nombre d'appels** faits par des clients : on parle de volumétrie d'appels
- Pouvoir retrouver tous les appels faits par des clients dans des **archives**.

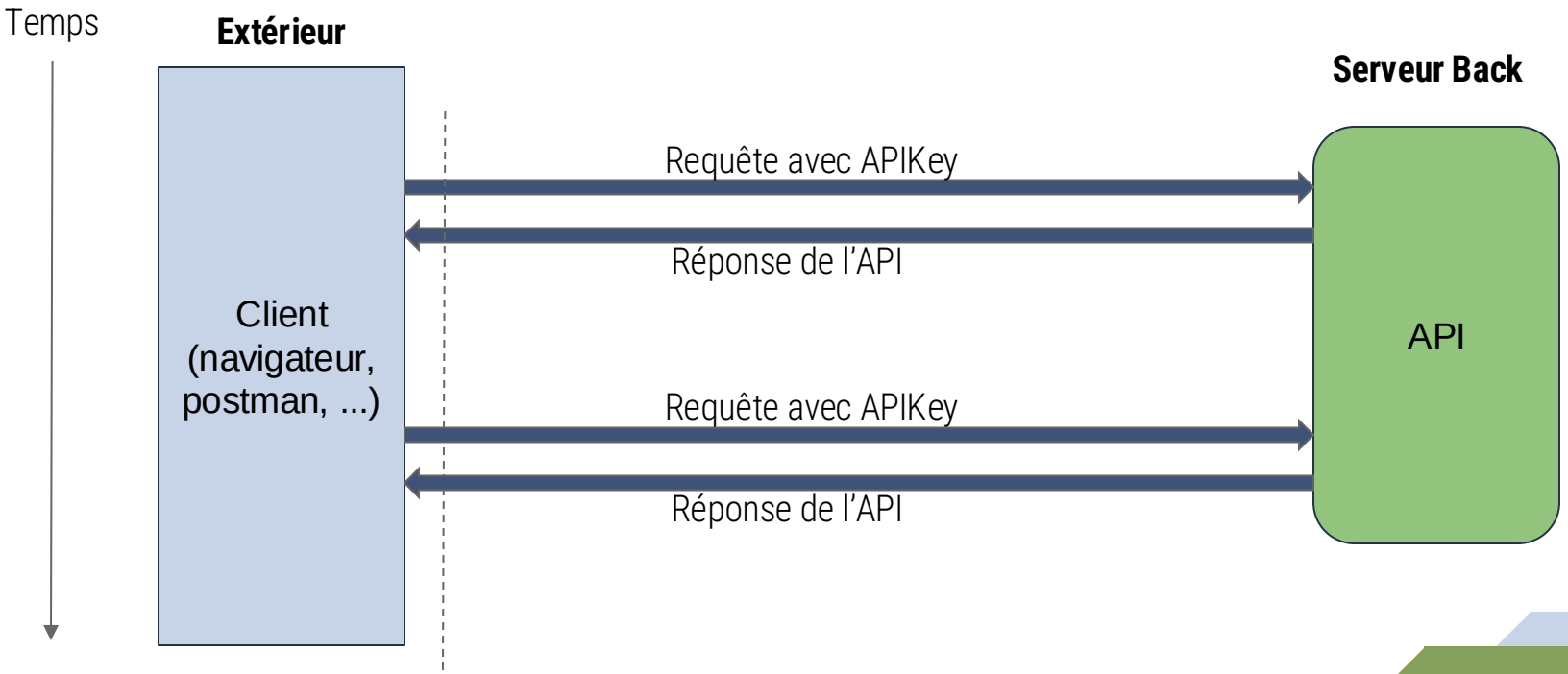
1/ API Key

Identification, pas authentification

Une API Key ne sert pas à :

- Identifier un utilisateur unique : Une API Key, est utilisée par une application complète, pas par un utilisateur spécifique.
- Gérer finement les droits : vu qu'elle est **partagée par de nombreux utilisateurs**.

1/ API Key


Temps
Extérieur
Serveur Back
Client
(navigateur,
postman, ...)
Requête avec APIKey
Réponse de l'API
API
Requête avec APIKey
Réponse de l'API

2/ Basic

Basic = simple

Cette fois-ci il est bien question d'authentification. L'utilisateur final de l'application ou de l'API va lui envoyer des informations personnelles d'identification et pas un compte partagé.

L'envoi de cette donnée est à faire dans un header **Authorization**.

Ce header sert spécifiquement à authentifier un utilisateur et sera réutilisé dans les autres solutions.

2/ Basic

Base 64

Le header Authorization doit toujours avoir la même structure :

Basic <identifiant:mot_de_passe en base 64>

Le **base64** est une autre représentation de la donnée, codée non pas en caractères classique, mais dans un autre **encodage**.

https://fr.wikipedia.org/wiki/Base64

Base 64

Il est très facile de passer du texte au base64 et inversement !

https://www.base64encode.org/ et https://www.base64decode.org/

En mathématiques, on dit que le base 64 est Bijectif : il ne peut pas y avoir de conflit. Il est également plus gourmand en espace que le stockage textuel.

Ce n'est pas un chiffrement sécurisé du tout !!

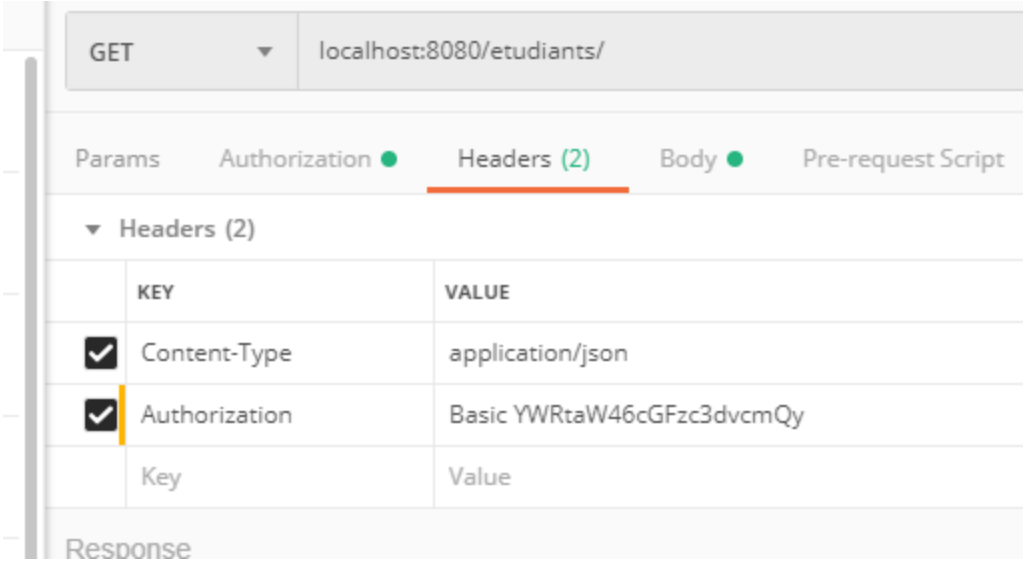
2/ Basic

Exemple

Ici le couple login:MDP est admin:password2.

En base 64 cela donne :

YWRtaW46cGFzc3dvcmQy



Serveur

Via des mécaniques internes, le serveur va vérifier que le couple login/mdp est correct et ensuite vérifier que l'utilisateur a bien les droits sur la ressource demandée.

Avec cette solution les mots de passe transitent **en clair** à chaque requête! Il faut donc idéalement :

- Chiffrer les mots de passe
- Utiliser HTTPS

HTTPS

Deux utilités :

- **S'assurer de la destination**. Elle est identifiée par un certificat émis par une autorité de certification reconnue.
- **Chiffrer le contenu des requêtes HTTPs**. Le certificat du serveur et un protocole de chiffrement compris par le serveur et le navigateur permettent de convertir les requêtes HTTP en un code uniquement compréhensible de l'émetteur et du récepteur.

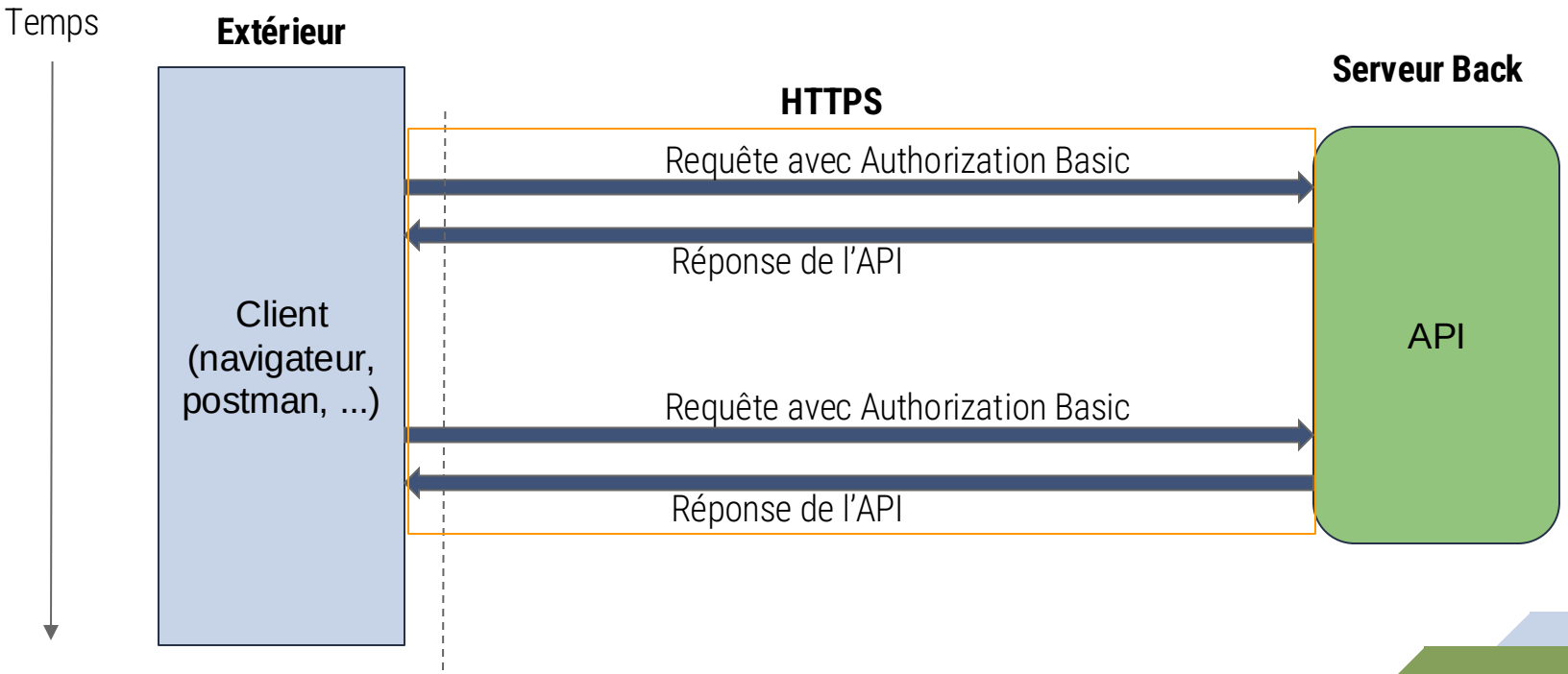
2/ Basic

HTTPS

“Serveur et client se sont reconnus, ont choisi une manière de chiffrer la communication et se sont passé de manière chiffrée un code pour dialoguer de manière secrète.” Wikipedia

C’est le b.a-ba de la sécurité sur le WEB. Toute communication extérieure un tant soit peu sensible **devrait** être en HTTPS.

2/ Basic


Temps
Extérieur
Serveur Back
HTTPS
Requête avec Authorization Basic
Réponse de l'API
Client (navigateur, postman, ...)
API
Requête avec Authorization Basic
Réponse de l'API

3/ Token

Un jeton d'authentification

Cette fois ci l'authentification se fait en deux temps :

1. Authentification sur l'API sur une ressource spécifique.Elle nous renvoie un **Jeton (Token)** si l'authentification est bonne, une erreur sinon.

1. Renvoi du jeton par le client pour chacune des requête vers l'API. Ce dernier ne contient pas l'identifiant ou le mot de passe. Il est d'usage de l'envoyer dans le header **Authorization**

3/ Token

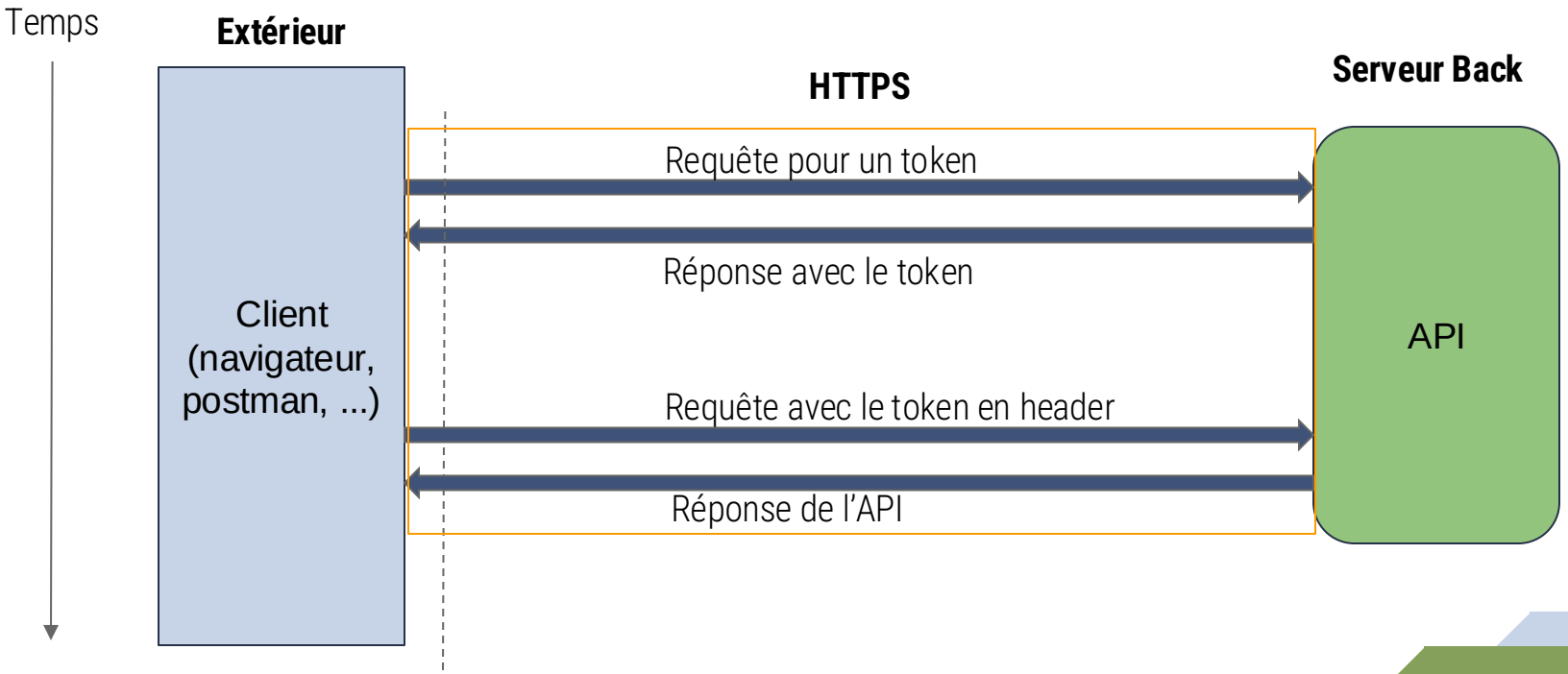
Une amélioration

Avec cette approche, le mot de passe ne transite qu'une fois sur le réseau, lors de l'authentification.

Généralement, une ressource /token ou /login existe et doit donc être appelé en POST. Quand un client déjà authentifié cherche à se reconnecter, on lui renvoie son token.

La durée de validité du token est paramétrable, et surtout il peut être **révoqué si nécessaire** (attaque d'un client donné).

3/ Token


Temps
Extérieur
HTTPS
Serveur Back
Client (navigateur, postman, ...)
Requête pour un token
Réponse avec le token
API
Requête avec le token en header
Réponse de l'API

4/ Serveur tiers

Le plus sécurisé, le modèle de référence actuel

L'approche est très similaire à celle du **Token** précédemment décrite.

La différence notoire ici est que le token est demandé et fourni par un **Serveur tiers d'authentification** dont la seule tâche est d'authentifier et de vérifier les droits.

Ce modèle est actuellement le plus sécurisé et donc celui **préconisé**.

Intégré à d'autres procédés

Ce modèle peut être enrichi par ou avec d'autres protocoles et outils afin de rajouter encore des niveaux de sécurité supplémentaires :

- Oauth2 : https://oauth.net/2/
- JWT : https://jwt.io/
- ...

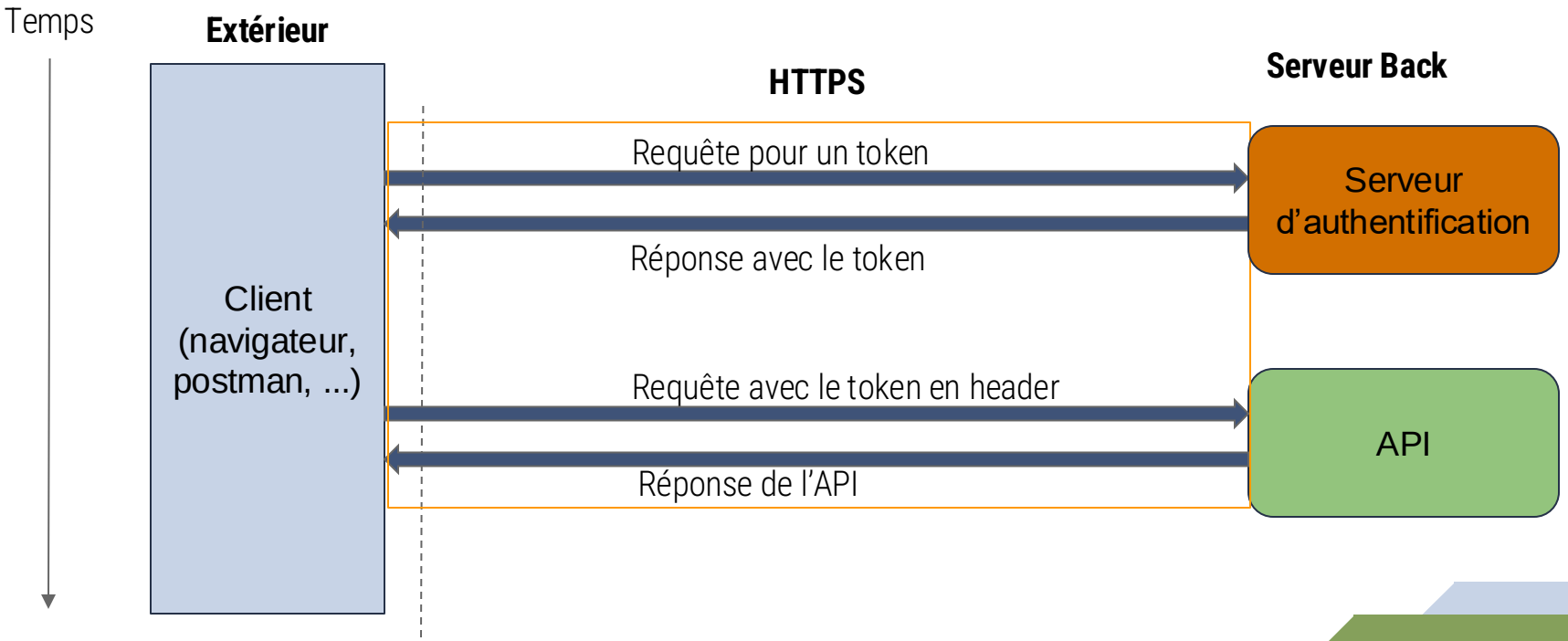
4/ Serveur tiers

Complexe et coûteuse

Toutefois cette méthode est celle nécessitant le plus de travail et le plus de configuration préalable.

Si la sécurité ne doit **jamais être négligée** , Oauth2 ou les serveurs d'authentification ne sont pas adaptés au développement d'une petite API utilisée par peu de personnes et qui ne serait pas exposée sur le Web.

4/ Serveur tiers



Sécuriser une API REST

Utiliser Spring Security

Librairie de référence

Spring Security

Faisant partie de la galaxie Spring, Spring Security est la librairie la plus utilisée dans le monde Java pour sécuriser les applications Web, y compris les APIs REST.

Comme l'ensemble du framework, sa configuration va s'appuyer sur des **Beans** issus du contexte et éventuellement modifiés dans des classes de `@Configuration`.

Dépendance

Spring Boot

En contexte Spring Boot, beaucoup de cas vont être gérés par **l'autoconfiguration**, rendant son usage bien plus simple que dans les applications Spring "classiques".

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

Basic

Sujet TRES complexe

Tout connaître des rouages de Spring Security est impossible pour un développeur dont ce n'est pas le métier.

Le nombre de classes intervenant dans chacun des processus (authentification, vérification d'un mot de passe, contrôle d'accès, …) est important.

La suite du cours ne s'attardera que sur la mise en place d'une authentification "Basic".

Activer la sécurisation

Activer la sécurisation d'une API Rest se fait via l'annotation `@EnableWebSecurity` à appliquer sur une classe de configuration OU sur la mainClass.

Via l'autoconfiguration, plusieurs beans seront ajoutés au contexte, prêts à être configurés.

Configuration

WebSecurityConfigurerAdapter

L'essentiel de la configuration de Spring Security se fait dans une classe de `@Configuration` **héritant de** `WebSecurityConfigurerAdapter`.

Hériter de cette classe force à redéfinir cette méthode :

```
protected void configure(HttpSecurity http) throws Exception {
}
```

Elle permet de définir quelle sécurité appliquer et quels rôles gérer.

Configuration

Exemple

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.csrf().disable()
            .authorizeRequests()
            .antMatchers(HttpMethod.POST, "/**").hasAnyRole("ADMIN")
            .anyRequest().permitAll()
            .and()
            .httpBasic();
}
```

Configuration

Exemple

Toute la configuration se fait en chaînant des méthodes sur l'objet de type HttpSecurity:

- **csrf().disable()** : Les attaques **C**ross-**S**ite **R**equest **F**orgery ne concernent que les sites Web. Elles peuvent être désactivés pour les API.

- **authorizeRequests()** est le marqueur de début du chaînage des contrôles d'accès, lien entre les ressources, les méthodes et les droits.

Configuration

- **antMatchers()** : Permet d'appliquer un contrôle sur une URL, une méthode HTTP, ou les 2. Un ou plusieurs **Rôles** deviennent ainsi nécessaires pour utiliser le combinaison ressource + méthode précédemment décrite.

  Les URLs peuvent contenir des caractères spéciaux, notamment **"*"** pour dire "n'importe quelle ressource", ou "**" pour "n'importe quelle URL".

Exemple :

`antMatchers(HttpMethod.POST, "/**").hasAnyRole("ADMIN")` : rôle ADMIN requis pour tous les appels POST de l'application.

Configuration

Exemple :

`antMatchers(HttpMethod.GET, "/resource").hasAnyRole("ADMIN")` : rôle ADMIN requis pour l'appel à /resource en GET.

`antMatchers(HttpMethod.POST, "/**").hasAnyRole("ADMIN","USER")` : rôle USER ou ADMIN requis pour tous les appels POST de l'application.

L'ordre de déclaration des antMatcher a donc une importance! La requête sera filtrée par le premier "Matchant".

Configuration

- **anyRequest()** : à placer derrière les antMatchers(). Il correspondra à toutes les requêtes non reconnues par un Matcher.

- **permitAll()** : revient à ne pas mettre de rôle prérequis pour une ressource ou une méthode. Il peut être utilisé à la place de hasAnyRole() sur un matcher.

- **and()** : permet simplement de chaîner et de récupérer à nouveau un objet de type HttpSecurity.

Configuration

- **httpBasic()** : active la sécurisation par Basic. Ceci permet à Spring de comprendre qu'il doit chercher un header **Authorization** et y recherche un couple login/mdp en base64.

Il est possible de remplacer httpBasic() par oauth2Client(), oauth2Login(), etc.

En mémoire

Pour des besoins très restreints, ou simplement le temps du développement, il est possible de définir des utilisateurs jetables, directement dans le code applicatif.

Pour cela il faut définir cette méthode dans la classe de configuration :

```
public void configureGlobal(AuthenticationManagerBuilder auth)
        throws Exception {}
```

Gestion des utilisateurs

Exemple

```
@Autowired
public void configureGlobal(AuthenticationManagerBuilder auth)
        throws Exception
{
   auth.inMemoryAuthentication()
            .withUser("admin")
            .password(passwordEncoder().encode("password2"))
            .roles("USER","ADMIN");
}
```

Plusieurs utilisateurs

Si plusieurs utilisateurs sont à créer par ce biais, il faut chaîner les appels avec la méthode `and()` qui retourne la même chose que `inMemoryAuthentication()`.

```
auth.inMemoryAuthentication()
        .withUser("admin")
                .and()
                .withUser("toto")
```

.

Bean obligatoires

Password encoder

Depuis ses versions les plus récentes, Spring security vous oblige à définir un objet de type PasswordEncoder : Il chiffre les mots de passes entrants.

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

Gestion des utilisateurs

Depuis une source de données

En règle général, les **utilisateurs** de l'application seront à charger depuis une **source de données extérieure** : base de données relationnelle, base NoSql, fichiers, annuaire LDAP, web service externe, etc.

Ceci remplace alors le chargement des utilisateurs en mémoire via la méthode configureGlobal.

Gestion des utilisateurs depuis une source externe

UserDetailsService

Une interface de Spring Security permet de faire lien entre la source de données et les beans utilisés dans la sécurisation : **UserDetailsService**. Elle est à implémenter dans l'application dans une classe qui sera chargée de faire cette passerelle.

Elle oblige à implémenter :

```
UserDetails loadUserByUsername(String var1) throws UsernameNotFoundException;
```

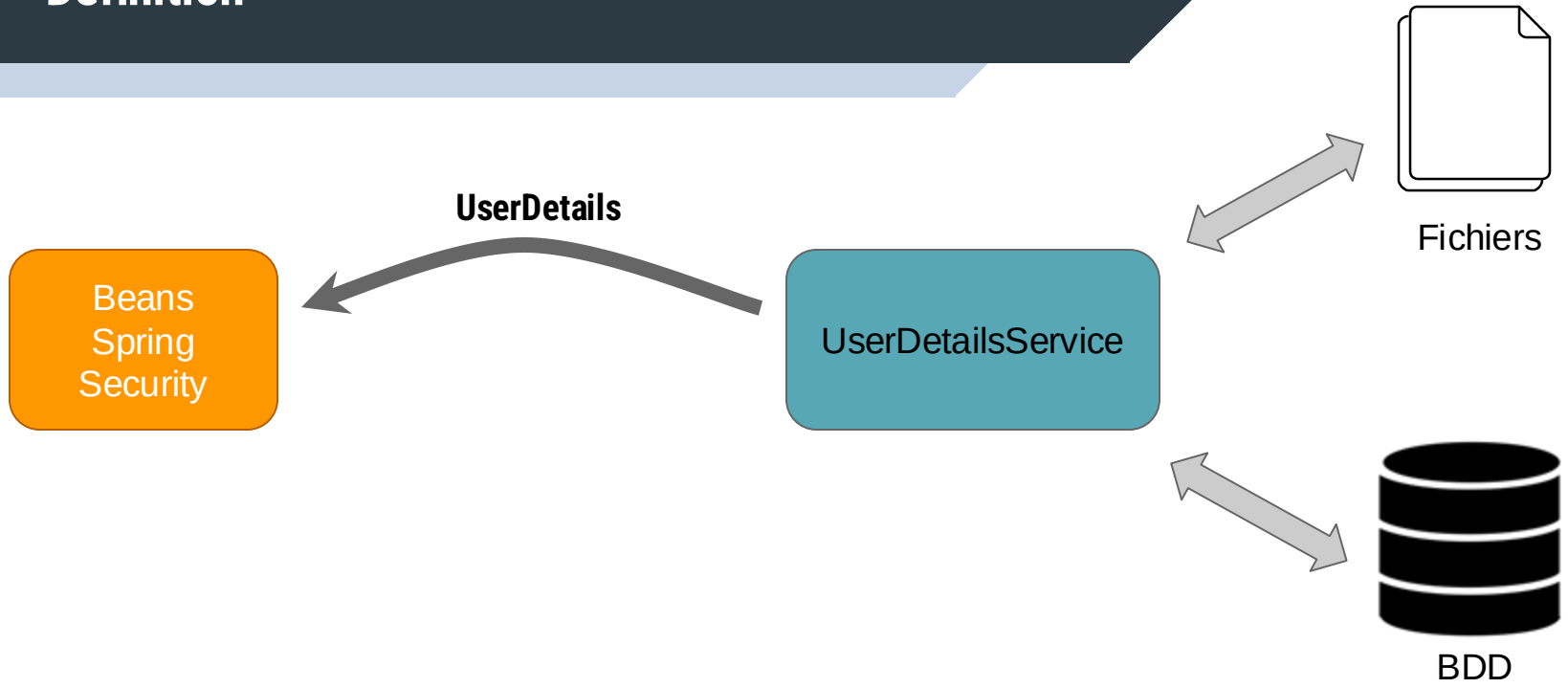
Gestion des utilisateurs depuis une source externe

UserDetails

UserDetails est la représentation en Objet d'un utilisateur (login + mot de passe) et de ses droits côté Spring Security.

La méthode `loadUserByUsername` est chargée d'extraire les données de la source externe, et de retourner un UserDetails bien formé avec ses informations.

Définition


UserDetails
Beans
Spring
Security
UserDetailsService
Fichiers
BDD

Création d'un UserDetails

```
@Override
public UserDetails loadUserByUsername(String username) {
PasswordEncoder encoder =
    PasswordEncoderFactories.createDelegatingPasswordEncoder();
Utilisateur user = userRepository.findByLogin(username)
      .orElseThrow(() -> new UsernameNotFoundException(username));
UserBuilder builder = User.withUsername(username);
builder.password(encoder.encode(user.getPassword()));
builder.roles(user.getRole());
return builder.build();
}
```

Création d'un UserDetails

La création d'un UserDetails peut se faire via un **`UserBuilder`** et la méthode `User.`*`withUsername`*`(String username)`

Au builder sont ensuite ajoutés:

- le mot de passe de l'utilisateur : **.password**(String password)
- les rôles : **.roles**(String … roles)

Le UserDetails est crée en appelée la méthode .build() sur le builder.

Chiffrement du mot de passe

Encodage du mot de passe

Ici encore, il est nécessaire d'utiliser un PasswordEncoder qui "chiffrera" le mot de passe envoyé.

```
PasswordEncoder encoder =
PasswordEncoderFactories.createDelegatingPasswordEncoder();

builder.password(encoder.encode(user.getPassword()))
```

Utilisateur inconnu

Exception

Si l'utilisateur est inconnu, il ne faut **pas retourner null**!

Il faut lever une `UsernameNotFoundException`.

Spring security interprétera correctement cette exception.