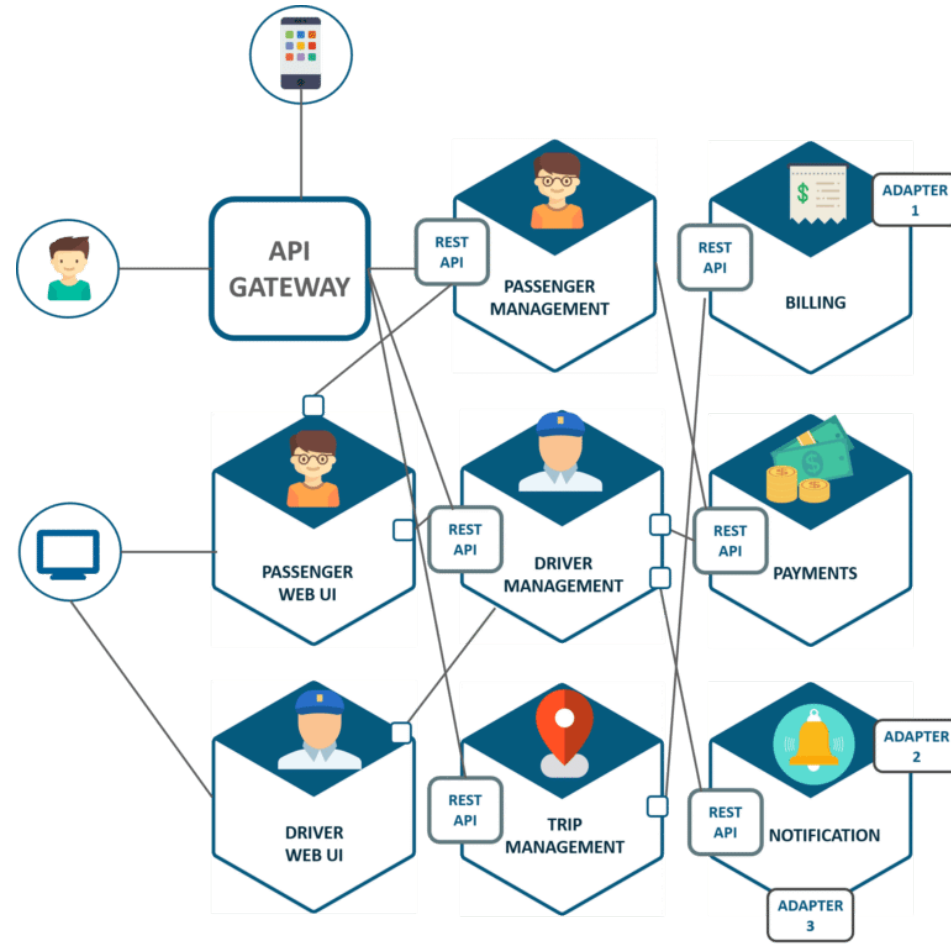


ALOM



SECURITY

UBER



PROBLÉMATIQUES :

- Comment sécuriser les données ?
- Comment authentifier les utilisateurs ?



NIVEAUX DE SÉCURITÉ

- Physique : Contrôle d'accès, biométrie
- Hardware : Encryption des disques
- Middleware : Firewalls (blocage d'IP/Ports), VPN (réseaux privés virtuels)
- Software : Authentification/Autorisation
- Data : Hashage / Chiffrement

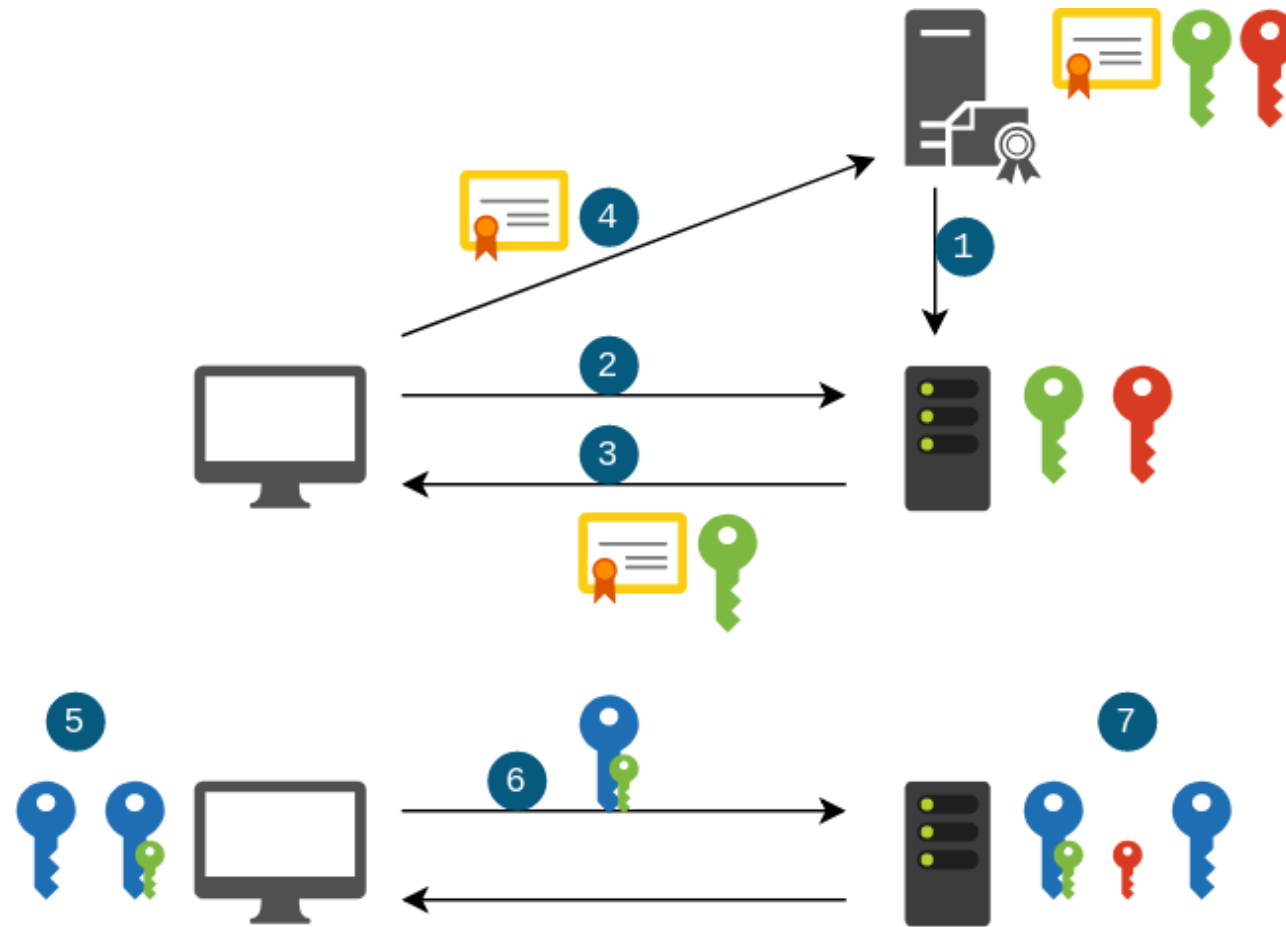
HTTPS

HTTPS fournit un tunnel de communications sécurisé

Encryption des données via un algorithme
asymétrique

Certificat validant l'identité du site + clé publique

HTTPS



HTTPS

- Chiffre les données entre le client et le serveur
- Ne permet pas de valider l'identité de l'utilisateur

SOFTWARE SECURITY

- Authentication (authentification)
- Authorization (autorisation)

AUTHENTICATION



Vérification de l'identité d'un "principal" (un user, un device, un système qui veut effectuer une action)

AUTHORIZATION



Décider si un "principal" peut faire une action en particulier. (contrôle d'accès)

AUTHENTIFICATION EN HTTP

Utilisation du header

```
Authorization: <type> <credentials>
```

```
Authorization: Basic QXNo0nBhc3N3b3Jk
```

```
Authorization: Bearer  
QXNo0nBhc3N3b3Jk
```

AUTHENTIFICATION EN HTTP

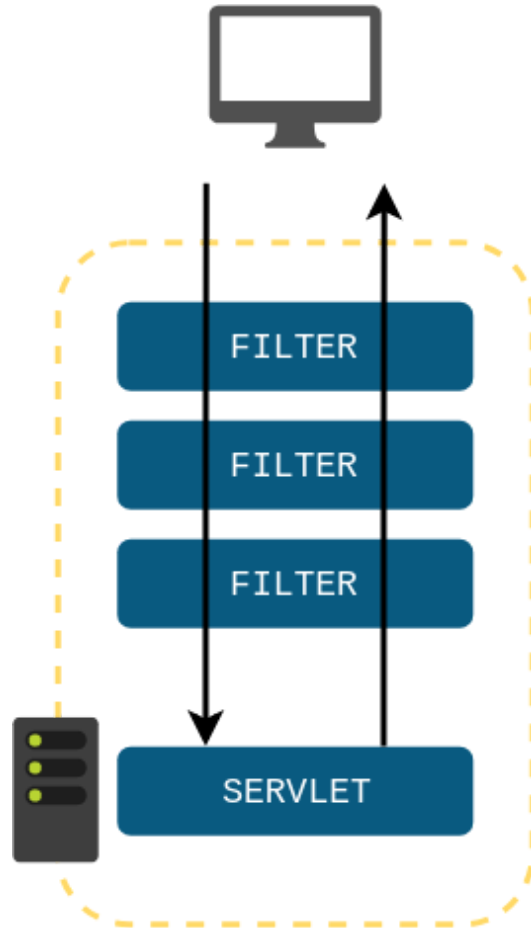
Authorization: Basic QXNo0nBhc3N3b3Jk

Les logins/mots de passe (ou tokens) transitent dans
les headers

C'est pour ça que l'on doit utiliser HTTPS !

EN SERVLETS

Utilisation des servlet filters



SPRING-SECURITY

- Authentication (validation des credentials)
- Utilisation d'un Cookie HTTP pour identifiant de session
- Stockage de "principal" en session côté serveur
- Logout : suppression de la session
- Protection contre le vol de session (CSRF & Session Fixation)
- Protection contre les appels venant de sources inconnues (CORS)

SPRING-SECURITY

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

Le simple fait d'ajouter spring security au classpath sécurise toutes les routes d'une application et ajoute une page de login par défaut.

SPRING-SECURITY

- ⚠ Les points listés dans ces slides sont pour Spring Boot 3 et Spring Security 6 (état de l'art)
- ⚠ Pour d'anciennes version de Spring Boot (2), les configuration sont différentes, mais les grands principes sont les mêmes

SPRING-SECURITY

2 *Beans* importants :

Un bean de type `SecurityFilterChain` permet de configurer les règles de sécurité :

- routes à protéger
- écran de login

Spring Security utilise des `AuthenticationProvider` pour valider les login/mdp ou tokens, et charger des rôles d'accès.

SPRING-SECURITY

SecurityFilterChain

```
@Configuration
public class SecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http)
        http
            .csrf(Customizer.withDefaults())
            .authorizeHttpRequests(authorize -> {
                authorize.requestMatchers("/unsecured/**").permitAll()
                authorize.requestMatchers("/api/**").authenticated()
                authorize.requestMatchers("/api/admin/**").hasRole("ADMIN")
                authorize.anyRequest().authenticated();
            })
            .httpBasic(Customizer.withDefaults())
            .formLogin(Customizer.withDefaults());
}
```

SPRING-SECURITY

AuthenticationProvider

Par défaut, Spring Security initialise un `DaoAuthenticationProvider`, qui s'appuie sur un `UserDetailsService`.

```
@Configuration
public class SecurityConfig {

    @Bean
    public UserDetailsService userDetailsService() {
        UserDetails userDetails = User.withDefaultPasswordEncoder
            .username("user")
            .password("password")
            .roles("USER")
            .build();

        return new InMemoryUserDetailsManager(userDetails);
    }
}
```

SPRING-SECURITY

Récupération des utilisateurs via un `UserDetailsService`. Par défaut, un `UserDetailsService` *in-memory* est créé à partir des propriétés:

- `spring.security.user.name`
- `spring.security.user.password`
- `spring.security.user.roles`

SPRING-SECURITY

Configuration d'un `UserDetailsService` avec un Bean Spring. Un `UserDetailsService` doit retourner des `UserDetails`.

```
@Autowired
private TrainerService trainerService;

@Bean
@Override
public UserDetailsService userDetailsService() {
    return username -> Optional.ofNullable(trainerService.getT
        .orElseThrow(() -> new BadCredentialsException("No
    }
}
```

SPRING-SECURITY

Page de login



```
1 <form action="/login" method="post">
2   <div class="form-group">
3     <label for="username">User Name : </label>
4     <input type="text" class="form-control" name="username"/>
5   </div>
6   <div class="form-group">
7     <label for="password">Password: </label>
8     <input type="password" class="form-control" name="password"/>
9   </div>
10  <input type="hidden" name="{{_csrf.parameterName}}" value="{{_csrf.token}}"/>
11  <div>
12    <input type="submit" class="btn btn-primary" value="Sign In"/>
13  </div>
14 </form>
```

SPRING-SECURITY

Accès à l'utilisateur loggué.

Injection du `java.security.Principal`

```
@GetMapping("/otherTrainers")
public Iterable<Trainers> getOtherTrainers(Principal principal) {
    return trainerRepository.findOtherTrainers(principal.getName());
}
```

`SecurityContextHolder`

```
@GetMapping("/otherTrainers")
public Iterable<Trainers> getOtherTrainers() {
    var auth = SecurityContextHolder.getContext().getAuthentication();
    var principal = (Principal) auth.getPrincipal();
    return trainerRepository.findOtherTrainers(principal.getName());
}
```

SPRING-SECURITY

Sécurisation des services REST par défaut

- Username : `user`
- Password : loggué sur la console `Using generated security password:`

`112eb169-1567-42fe-bf0e-7c7bc94a5afa`

SPRING-SECURITY

Personnalisation de la sécurisation des services REST

- Username : `spring.security.user.name`
- Password :
`spring.security.user.password`

SPRING-SECURITY

GESTION DES RÔLES.

Un user a des `GrantedAuthority`.

Une `GrantedAuthority` = un *rôle*.

A positionner par les `AuthenticationProvider`,
ou par le `UserDetailsService`, dans les objets
`UserDetails`.

```
UserDetails userDetails = User.withDefaultPasswordEncoder()  
    .username("user")  
    .password("password")  
    .roles("USER") // ici ! crée un GrantedAuthority()  
    .build();
```

SPRING-SECURITY

GESTION DES RÔLES.

Autorisations au niveau des méthodes (contrôleur, ou service).

ANNOTATIONS JSR-250 (STANDARD JAVA, DANS `jakarta.annotation-api`)

```
@Configuration
@EnableMethodSecurity(jsr250Enabled = true)
public class MethodSecurityConfig {
}
```

```
@RolesAllowed("TRAINER") // sans le "ROLE_"
ModelAndView myTrainerMethod(){...}
```

```
@RolesAllowed("ADMIN")
ModelAndView myAdminMethod(){...}
```



SPRING-SECURITY

GESTION DES RÔLES.

Autorisations au niveau des méthodes (contrôleur, ou service).

ANNOTATIONS SPRING SECURITY [DOC](#)

`@PreAuthorize`, `@PostAuthorize`,
`@PreFilter`, `@PostFilter`.

```
@Configuration
@EnableMethodSecurity
public class MethodSecurityConfig {
}
```

```
@PreAuthorize("hasRole('TRAINER')") // sans le "ROLE_"
ModelAndView myTrainerMethod(){...}
```

```
@PostAuthorize("returnObject.name == authentication.name")
```



RECOMMANDATION GÉNÉRALES

Kids, don't do this at home

- Ne stockez **JAMAIS** de mot de passe en clair
- Ne faites **JAMAIS** de concaténation de chaînes pour générer des requêtes SQL
- Évitez de stocker des *secrets* dans des propriétés (utilisez des variables d'environnement)
- Utilisez du HTTPS et du TLS dès que possible
- Sécurisez vos routes HTTP, utilisez les annotations et les rôles
- Chiffrez les données en base si les données sont sensibles

TP



Security 