07/10/2023
Version 2

SPRING CLOUD

JÉRÉMY PERROUAULT

SÉCURITÉ

UN POINT D'ACCÈS UNIQUE

SÉCURITÉ

- Mettre en place
  - Une authentification utilisateur (`Authentication`)
  - Une validation des autorisations de l'utilisateur (`Authorization`)

- On peut implémenter une sécurité sur chaque service
  - Mais puisqu'on utilise une **Gateway** et que tous les flux passent par là, autant utiliser la **Gateway**

JWT

LES JETONS JSON

JWT

- Le protocole **HTTP**(**S**) est un protocole dit « déconnecté »
  - Chaque requête est indépendante
  - Il n'y a, par défaut, aucune stratégie de cohérence d'utilisateur

- Pour reconnaitre un utilisateur
  - Utilisation des sessions (côté serveur)
  - Utilisation des cookies (côté client – navigateur Web par exemple)
  - Les cookies sont envoyés à chaque requête, par le client, dans les en-têtes **HTTP**
    - **Cookie**
    - Sensible aux attaques **CSRF**

JWT

- **J**SON **W**eb **T**oken
  - Jeton généré par un serveur d'authentification, transmis au client
  - Remplace les sessions et cookies
  - Envoyés à chaque requête, dans les en-têtes **HTTP**
    - Authorization, avec le type d'autorisation « Bearer »
    - Il est aussi possible de le stocker et l'envoyer en tant que Cookie, mais attention aux attaques CSRF !

- Contrairement au « Cookie de session », le jeton peut contenir des informations
  - Elles sont signées à l'aide d'une clé privée détenue par le serveur
  - Lorsque le serveur reçoit un jeton, il compare la signature pour vérifier la validité

JWT

- Un jeton **JWT** est composé
  - `Header`
    - Contenant l'algorithme utilisé pour la signature et le type de jeton (**JWT**)
    - Au format **JSON**, encodé en **Base64**
  - `Payload`
    - Contenant les informations du jeton (nom d'utilisateur, scope, date d'émission, date d'expiration, etc.)
    - Au format **JSON**, encodé en **Base64**
  - `Signature`
    - Concaténation de `Header` et `Payload`, chiffrée avec la clé privée
    - Empêche donc l'altération des données

https://jwt.ms/

JWT

- Exemple Header **JWT**

```
{
  "alg":"HS512",
  "typ":"JWT"
}
```

- Exemple Payload **JWT**

```
{
  "iat": 1669658768,
  "exp": 1669662368,
  "sub": "jeremy",
  "scope": "produit.read"
}
```

JWT

- Exemple **JWT**

```
eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJqZXJlbXkiLCJzY29wZSI6InByb2R1aXQucmVhZCIsImlhdCI6MTY2OTY1ODc2OCwiZXhwIjoxNjY5NjYyMzY4fQ.r46SL8XB43Wzxql6ZvcpsAhvuxwLnEhMqeKZRvq86MFxffsibxBHq9m9jwv47iDCPel5xvhC33h0-6OcmNITLA
```

- Header
- Payload
- Signature

https://jwt.io/

JWT

- Pour utiliser le jeton, on ajoute « Authorization » à l'en-tête HTTP
- `Authorization` `Bearer Token`

- « Bearer » fait référence au type d'authorization spécifique
  - Souvent utilisé pour les jetons d'accès de ce type
- Il en existe d'autres
  - Basic — Authentification **HTTP-Basic** en **Base64**
  - Digest — Authentification **HTTP-Basic** en **MD5**
  - Negociate — Kerberos pour systèmes MS Windows
  - AWS4-HMAC-SHA256 — Pour AWS

JWT – TESTER

- Manipuler un Jeton
  - Consulter le jeton **Azure** ou **Google**
    - jwt.io ou jwt.ms
  - Fabriquer ou modifier un jeton
    - jwt.io

- Et en effet, le `header` et le `payload` sont en clair

JWT – EXERCICE

- Inclure les dépendances
  - **jjwt-jackson** & **jjwt-impl** (de **io.jsonwebtoken**)
  - Générer une clé SHA512
  - Utiliser un temps d'expiration de 3600000 ms

```
public String generateJwtToken(Authentication authentication) {
  SecretKey key = Keys.hmacShaKeyFor(this.jwtSecret.getBytes(StandardCharsets.UTF_8));

  return Jwts.builder()
    .setSubject(authentication.getName())
    .setIssuedAt(new Date())
    .setExpiration(new Date((new Date()).getTime() + this.jwtExpirationMs))
    .signWith(key)
    .compact();
}
```

JWT – EXERCICE

- Ajouter un filtre pour intercepter l'en-tête HTTP `Authorization`
  - Le jeton sera placé après `Bearer`
  - Retrouver l'utilisateur, y associer le ou les autorisations
  - Désactiver la protection **CSRF**
    - Qui n'est plus nécessaire car un jeton est demandé à chaque requête
  - Désactiver le stockage de la session utilisateur
    - L'utilisateur sera retrouvé grâce au jeton

OAUTH2

FOURNISSEURS OIDC

OAUTH2

- Problématique : application de composition d'album photo
  - L'utilisateur peut charger une image depuis son poste (pas de problème ici)
  - L'utilisateur peut vouloir partager des photos enregistrées sur un Cloud
    - L'application peut demander à l'utilisateur propriétaire ses identifiants (login / password)
    - Et **jurer** qu'elle ne gardera rien en mémoire et ne fera que ce qu'elle est sensée faire : charger une image

- Avant l'arrivée de **OAuth**
  - Les identifiants étaient demandées (et souvent stockés en clair !)
  - L'application avait accès à tout ce qu'avait accès l'utilisateur propriétaire

OAUTH2

- **OAuth2** pour **O**pen **Auth 2**.0
  - Comment une application peut-elle accéder à une ressource protégée, au nom de son propriétaire, sans connaitre ses identifiants (login / mot de passe) ?

- **OAuth2**, contrairement à la première version, **doit** s'exécuter en **HTTPS**
  - L'aspect de sécurité pour la confidentialité des données est délégué au protocole TLS

OAUTH2

- On distingue l'authentification et l'autorisation
- `Authentification`
  - Reconnaissance d'un utilisateur via ses identifiants de connexion
- `Autorisation`
  - Accès (ou non) à une ressource ou un ensemble de ressources
  - Exemple : un administrateur a plus d'autorisations qu'un utilisateur lambda

OAUTH2

- De façon classique, un processus d'authentification et d'autorisation
  - Un serveur, en mesure d'authentifier et d'autoriser l'accès à une ressource
  - Un utilisateur, en mesure de fournir ses identifiants
- Pour **OAuth**, le processus est un peu plus complexe
  - Propriétaire de la ressource **Resource owner**
    - Une entité (utilisateur par exemple) en mesure de donner l'accès à une ressource protégée
  - Client **Client**
    - Une entité (application ou site web) qui demande l'accès à une ressource protégée
  - Serveur de la ressource **Resource server**
    - Le serveur qui héberge la ressource protégée
  - Serveur d'autorisation **Authorization server**
    - Le serveur qui délivre le droit d'accès à la ressource protégée au client après avoir authentifié le propriétaire

OAUTH2

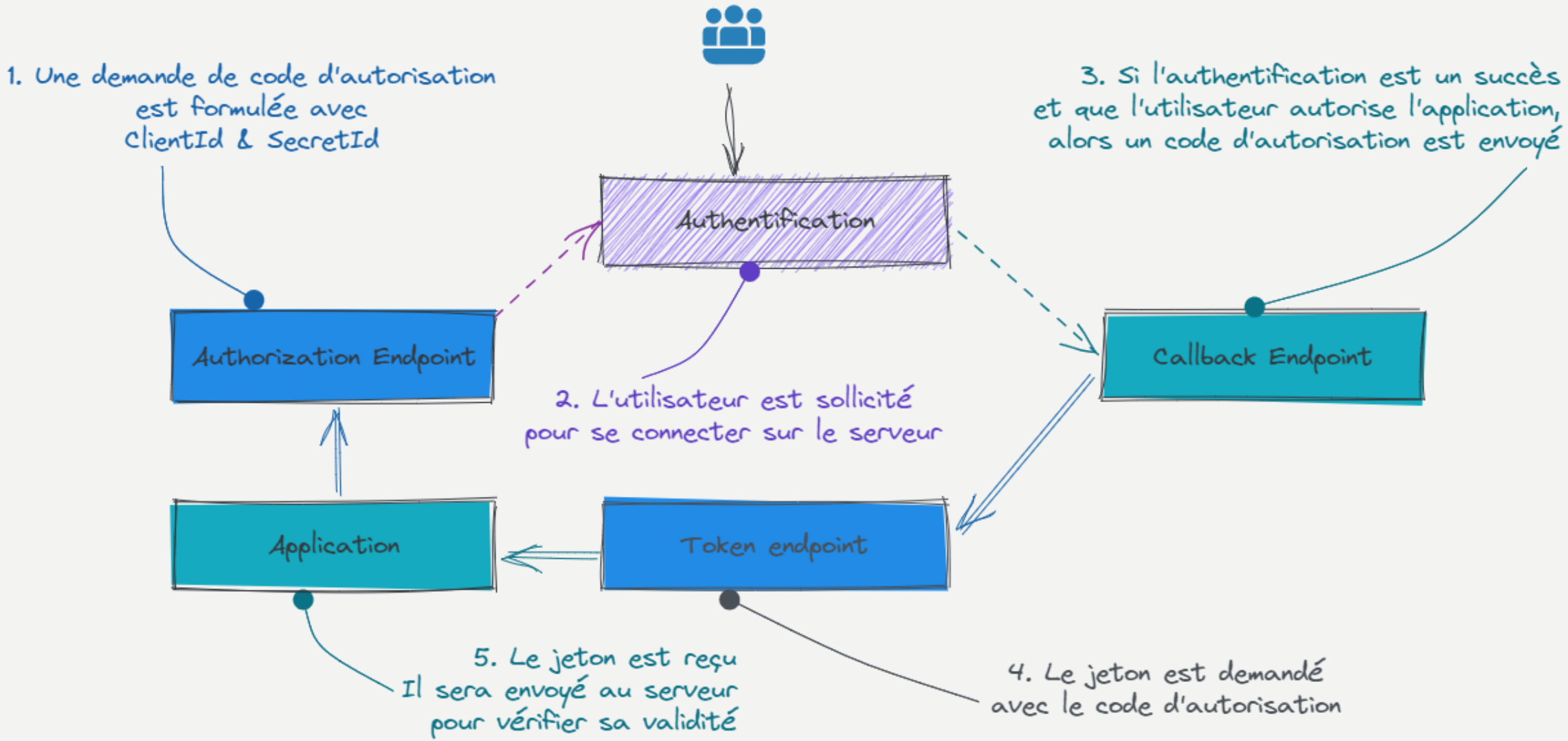
- La demande d'autorisation est toujours initiée par le client
  - Qu'il faut enregistrer auprès du serveur d'autorisation
- Son enregistrement nécessite
  - Un identifiant, l'identifiant du client
  - Un mot de passe, le secret du client
  - Une ou plusieurs URL de redirection, pour indiquer au client l'état de l'autorisation

OAUTH2

- La demande du client se traduit par la délivrance d'un jeton (token)
- Il en existe 2 types
  - Jeton d'accès **`Access Token`**
    - Jeton qui permet d'accès à une ressource protégée
    - Durée de validité
    - Peut avoir une portée (scope) limitée
      - Par exemple : lecture seule sur les photos uniquement
  - Jeton de rafraîchissement **`Refresh Token`**
    - Jeton qui permet d'obtenir un nouveau jeton d'accès sans l'intervention du propriétaire
    - Durée de validité, plus longue que le jeton d'accès

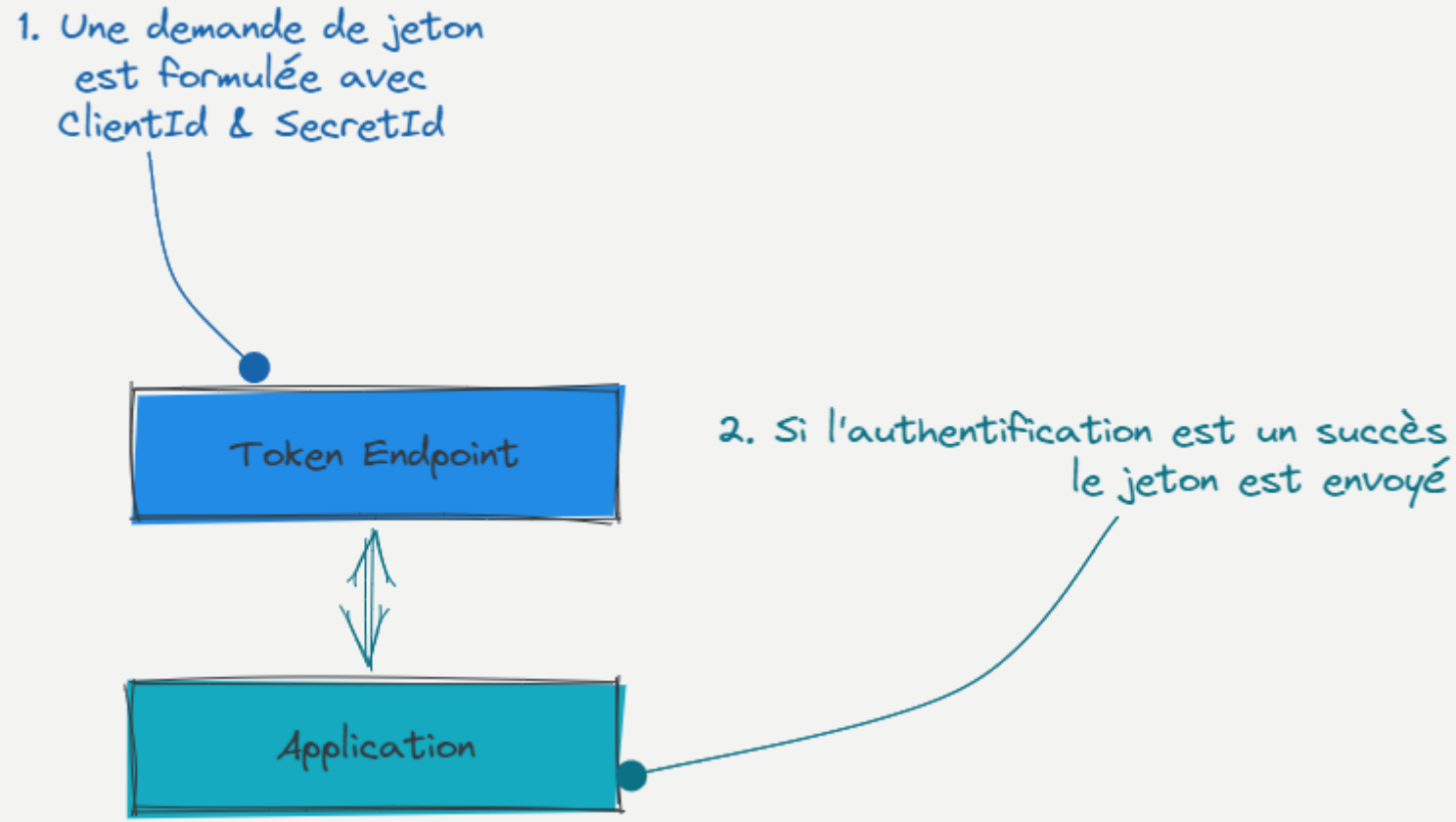
OAUTH2

- Type de demande : `Authorization Code`



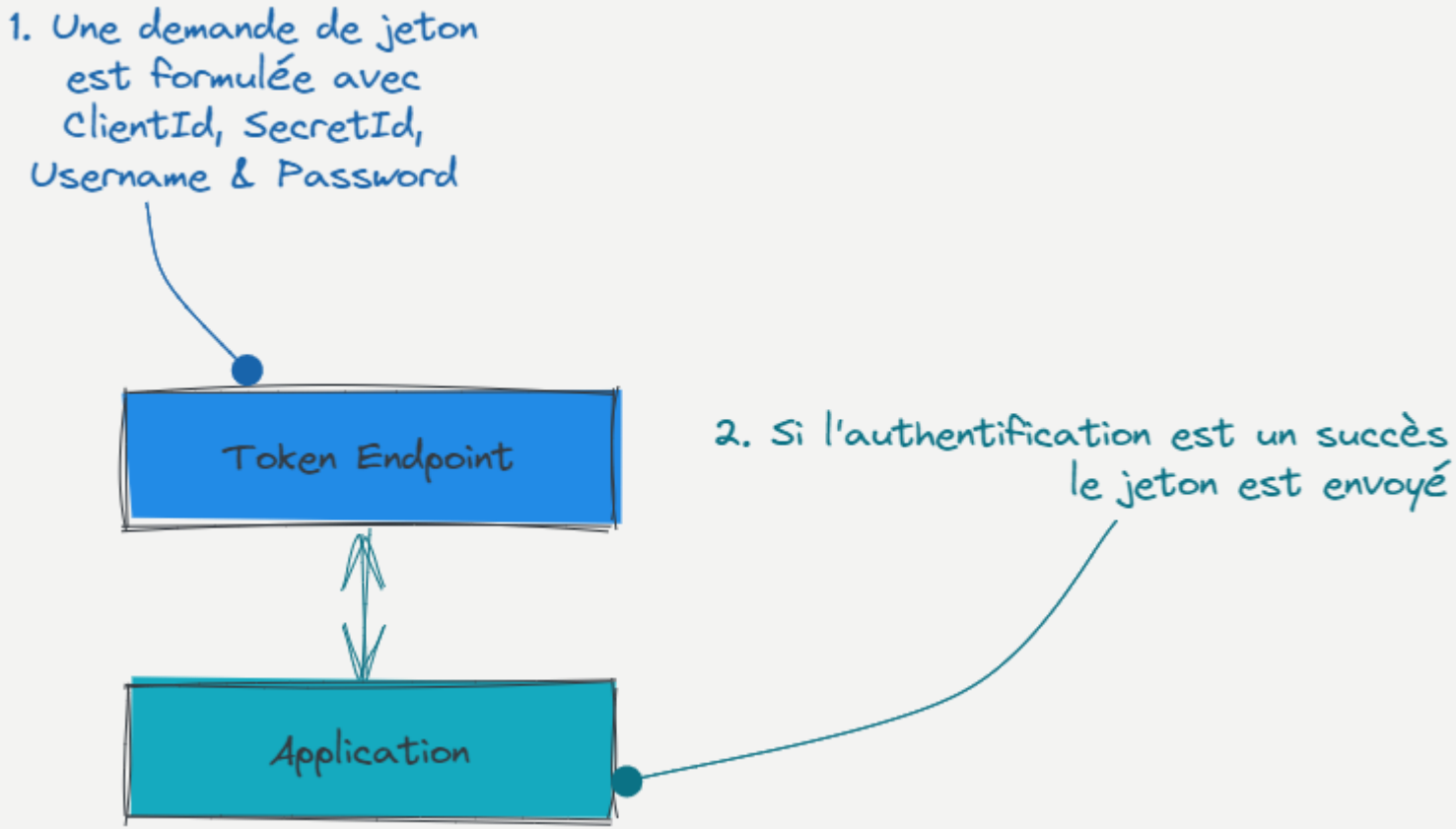
OAUTH2

- Type de demande : `Client Credentials`



OAUTH2

- Type de demande : `Password Credentials`



OAUTH2 – TESTER

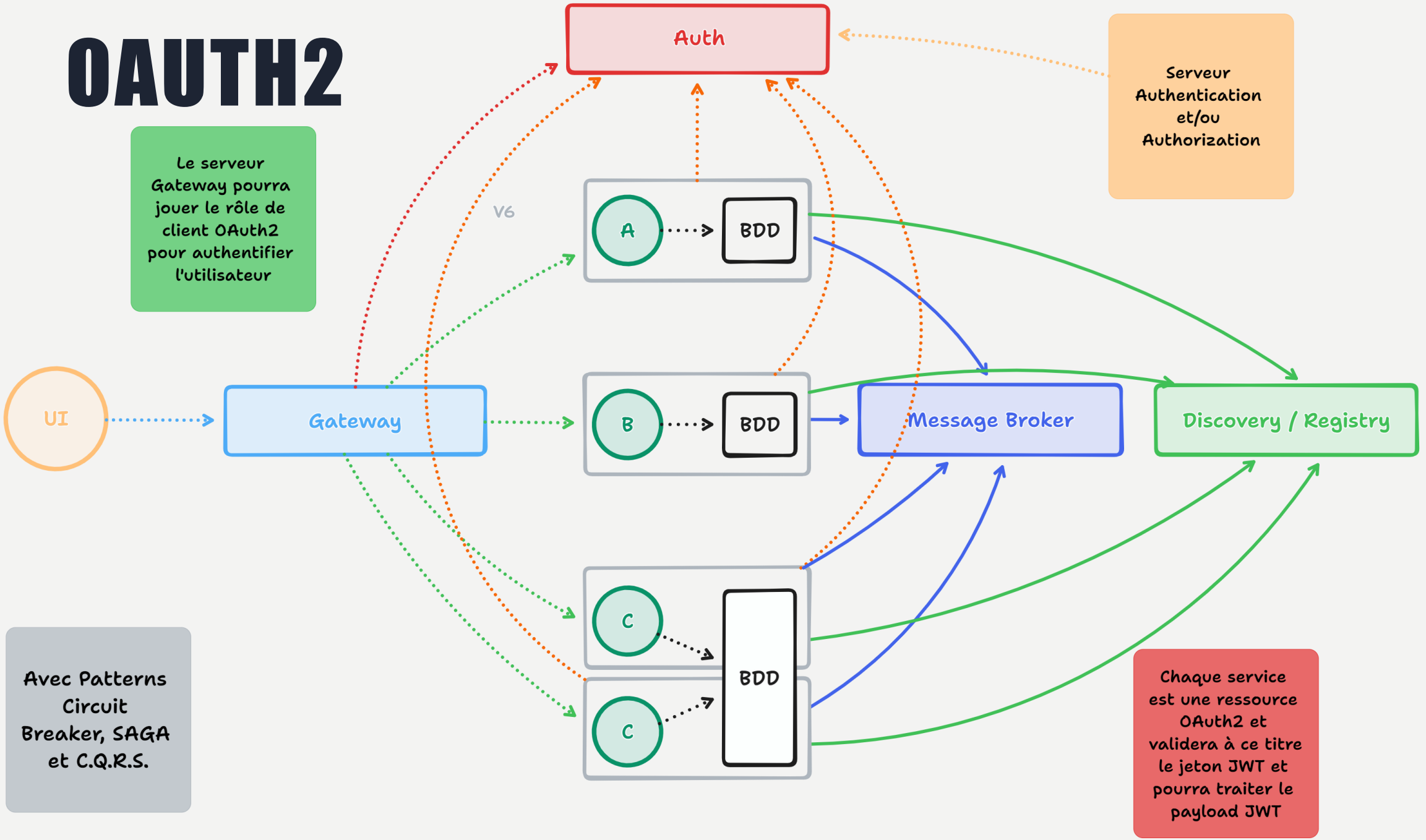
- Tester avec Postman une connexion **OAuth2 Azure** ou **Google**
  - portal.azure.com
  - console.cloud.google.com

OAUTH2

- **O**pen**ID** **C**onnect
  - Surcouche à **OAuth2**
  - Plus ouvert
  - Utilise les jetons d'accès au format **JWT**
    - Permet d'obtenir des informations sur l'utilisateur, sans en faire une demande spéciale (ces informations peuvent être stockées dans le jeton – ces informations sont en clair, et la clé permet de garantir que les données n'ont pas été altérées)

OAUTH2

Le serveur Gateway pourra jouer le rôle de client OAuth2 pour authentifier l'utilisateur



Avec Patterns Circuit Breaker, SAGA et C.Q.R.S.

Chaque service est une ressource OAuth2 et validera à ce titre le jeton JWT et pourra traiter le payload JWT

OAUTH2 – RESOURCESERVER

- Chaque **service** deviendra un serveur de ressource (et un relais de vérification d'un jeton)
  - Utiliser le starter **spring-boot-starter-oauth2-resource-server**
  - Activer le login via le serveur de ressource oauth2

```
http.oauth2ResourceServer(oauth2 -> oauth2.jwt(Customizer.withDefaults()));
```

  - Configurer les « relais » **JWT**

```
spring.security.oauth2.resourceserver.jwt.issuer-uri = http://localhost:8500/realms/demoformation
```

OAUTH2 – RESOURCESERVER

- Au besoin
  - On mappe les rôles qui se trouvent dans le `Payload` du **JWT** (`Claims`)
  - L'exemple qui suit est un exemple avec un jeton **JWT** de **KeyCloak**

OAUTH2 – RESOURCESERVER

- Pour mapper les autorisations d'un utilisateur **OIDC**
  - On utilise un `Converter`

```
private Converter<Jwt, Collection<GrantedAuthority>> jwtConverter() {
  return jwt -> {
    Map<String, Object> resourceAccess = jwt.getClaim("resource_access");
    Map<String, Object> demoClient = (Map<String, Object>) resourceAccess.get("demo-client");
    List<String> roles = (List<String>) demoClient.get("roles");
    List<GrantedAuthority> mappedAuthorities = new ArrayList<>();

    mappedAuthorities.addAll(roles.stream()
      .map(r -> new SimpleGrantedAuthority("ROLE_" + r))
      .toList()
    );

    return mappedAuthorities;
  };
}
```

OAUTH2 – RESOURCESERVER

- Converter utilisé par un JwtAuthenticationConverter

```
private JwtAuthenticationConverter jwtAuthenticationConverter() {
  JwtAuthenticationConverter jwtAuthenticationConverter = new JwtAuthenticationConverter();
  jwtAuthenticationConverter.setJwtGrantedAuthoritiesConverter(this.jwtConverter());

  return jwtAuthenticationConverter;
}
```

- JwtAuthenticationConverter utilisé dans la configuration du serveur de ressource

```
http.oauth2ResourceServer(server ->
  server.jwt(jwt -> jwt.jwtAuthenticationConverter(this.jwtAuthenticationConverter()))
);
```

OAUTH2 – CLIENT

- Utiliser le starter **spring-boot-starter-oauth2-client**
  - Configurer l'id et le secret de l'application (et éventuellement d'autres paramètres)

```
spring.security.oauth2.client.registration.google.client-id = XXX
spring.security.oauth2.client.registration.google.client-secret = XXX
```

OAUTH2 – CLIENT

- Configuration de la sécurité

```
@Bean
SecurityWebFilterChain securityFilterChain(ServerHttpSecurity http) throws Exception {
  http.oauth2ResourceServer(server ->
   server.jwt(jwt -> Customizer.withDefaults())
  );

  http.csrf(Customizer.withDefaults());

  return http.build();
}
```

NOTE Spring Security utilise par défaut la session pour stocker les jetons de l'utilisateur connecté

OAUTH2 – CLIENT

- Configuration de la sécurité

```
@Bean
SecurityWebFilterChain securityFilterChain(ServerHttpSecurity http) throws Exception {
  http.oauth2ResourceServer(server ->
   server.jwt(jwt -> Customizer.withDefaults())
  );

  http.csrf(Customizer.withDefaults());

  return http.build();
}
```

OAUTH2 – CLIENT

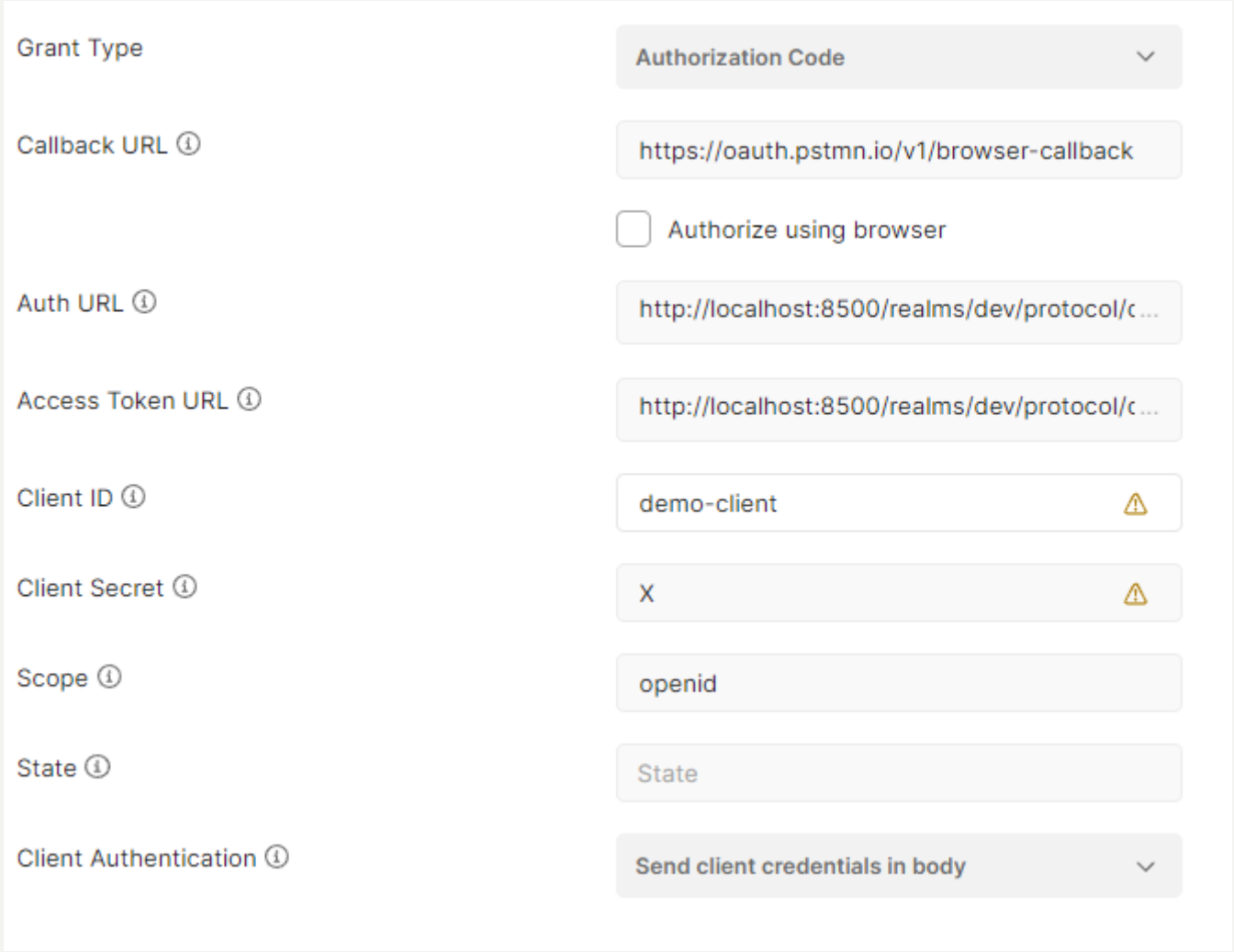
- Exemple de configuration plus complète pour une authentification plus spécifique

```
#OAuth2 Spécifique
spring.security.oauth2.client.registration.formation.client-name = Formation
spring.security.oauth2.client.registration.formation.client-id = XXX
spring.security.oauth2.client.registration.formation.client-secret = XXX
spring.security.oauth2.client.registration.formation.authorization-grant-type = authorization_code
spring.security.oauth2.client.registration.formation.redirect-uri = http://localhost:8500/login/oauth2/code/formation

spring.security.oauth2.client.provider.formation.token-uri = endpoint pour récupérer un jeton (avec un code d'autorisation par ex.)
spring.security.oauth2.client.provider.formation.authorization-uri = endpoint pour récupérer un code d'autorisation
spring.security.oauth2.client.provider.formation.user-info-uri = endpoint pour récupérer les infos user
spring.security.oauth2.client.provider.formation.user-name-attribute = nom attribut pour le nom de l'user
```

OAUTH2 – CLIENT

- Exemple avec **Postman**



EXERCICE

- Implémenter une authentification par **KeyCloak**