Mini discord

Spécification, Sécurité, Mise en oeuvre

# Spécifications :

Pour créer l’API, nous avons décider de la créer sur AWS (Amazon Web Service).

En effet AWS présente de nombreux avantages :

* Sa rapidité de mise à disposition des ressources de développement.
* AWS gère les ressources et le dimensionnement.
* Le premier million de requêtes est gratuit

Nous utiliserons Swagger (OpenAPI 3.0) pour réaliser nos tests, specs d’API.

Le format normé de réponse que nous avons choisi est : JSON

Nous nous sommes tournés vers ce choix principalement car en plus d’être le format le plus utilisé ainsi que celui que nous maitrisons le mieux, de plus :

* Format compréhensible par tous
* Ne dépend d'aucun langage
* Permet de stocker des données de différents type
* Sa structure et sa syntaxe simple lui permet de rester très "léger" et efficace.

#### Attention :

* Comme pour tout moyen de stockage de données, il faudra prendre des mesures de sécurité pour garantir la sécurité des données sensibles.

Notre API respecte le niveau 3 du modèle de Maturité de Richardson:

* Utilisation des ressources (Chaque ressource est atteignable via une URI spécifique.)
* Utilisation des verbes et codes http (GET, POST, PUT, DELETE, PATCH //

4XX : erreur côté client, …)

* HATEOAS (Liens hypermédias)

# EndPoints :

L’api est divisée en deux parties principales : COGNITO ET SERVERS.

COGNITO s’occupe de toute la partie utilisateurs : Inscription, Connection, Roles

SERVERS s’occupe de gérer les différents servers et leurs contenu (canaux, mesages)

## /Cognito :

{ressource} doit être remplacé par « login », « register » ou « admin », ceci va permettre de se connecter, s’inscrire ou de changer le rôle d’un utilisateur. Cette ressource possède 2 méthodes : OPTIONS et POST, lorsqu’un login ou un register est demandé, le body doit contenir un username et un password. Lorsque admin est spécifié, le body doit contenir l’email de l’utilisateur qui va avoir ses droits modifiés.

## /Servers :

Cette ressource possède 3 methodes : OPTIONS, POST, GET.

* GET : elle permet à l’utilisateur d’obtenir les serveurs auxquels il est enregistré, et si il est administrateur, il peut accéder à tous les serveurs
* POST : elle permet à l’utilisateur de créer un nouveau serveur en spécifiant dans le body le nom du serveur voulu, body : {server\_name : nomduserveur)
* OPTIONS : permet de valider la préflight request

## /Servers/{serverid} :

Cette ressource possède 4 methodes : OPTIONS, PATCH, GET, DELETE.

{serverid} correspond à l’id du serveur

* GET : permet à l’utilisateur d’obtenir les informations du serveur
* PATCH : permet à l’utilisateur de rejoindre ou quitter un serveur body : {action :leave ou add, user : test@test.com)
* DELETE : permet à un utilisateur de supprimer un serveur qu’il à créé
* OPTIONS : permet de valider la préflight request

## /Servers/{serverid} /channels :

Cette ressource possède 3 methodes : OPTIONS, POST, GET.

{serverid} correspond à l’id du serveur

* GET : permet d’obtenir les canaux du serveur passé en parametre
* POST : permet de créer un canal body : {name : nomducanal)
* OPTIONS : permet de valider la préflight request

## /Servers/{serverid} /channels/{channelid}

Cette ressource possède 3 methodes : OPTIONS, DELETE, GET.

{serverid} correspond à l’id du serveur

{ channelid } correspond à l’id du canal

* GET : permet d’obtenir le canal du serveur passé en paramètre
* DELETE : permet de supprimer le canal passé en parametre
* OPTIONS : permet de valider la préflight request

## /Servers/{serverid} /channels/{channelid}/messages

Cette ressource possède 3 methodes : OPTIONS, POST, GET.

{serverid} correspond à l’id du serveur

{ channelid } correspond à l’id du canal

* GET : permet d’obtenir tous les messages présents dans le canal du serveur
* POST : permet de créer un nouveau message dans le canal du serveur, body : {text : message)
* OPTIONS : permet de valider la préflight request

## /Servers/{serverid} /channels/{channelid}/messages/{messageid}

Cette ressource possède 3 methodes : OPTIONS, DELETE, GET.

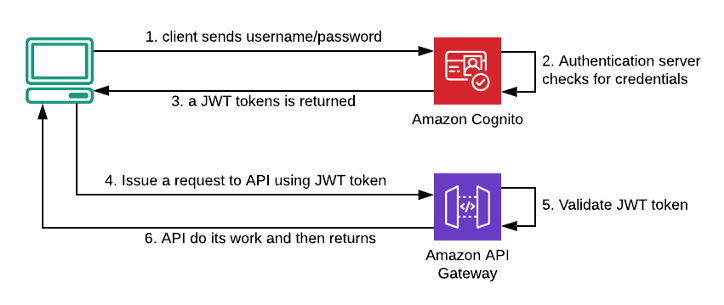
{serverid} correspond à l’id du serveur

{ channelid } correspond à l’id du canal

{ messageid } correspond à l’id du message

* GET : permet d’obtenir les informations du message
* DELETE : permet de supprimer le message dans le canal du serveur
* OPTIONS : permet de valider la préflight request

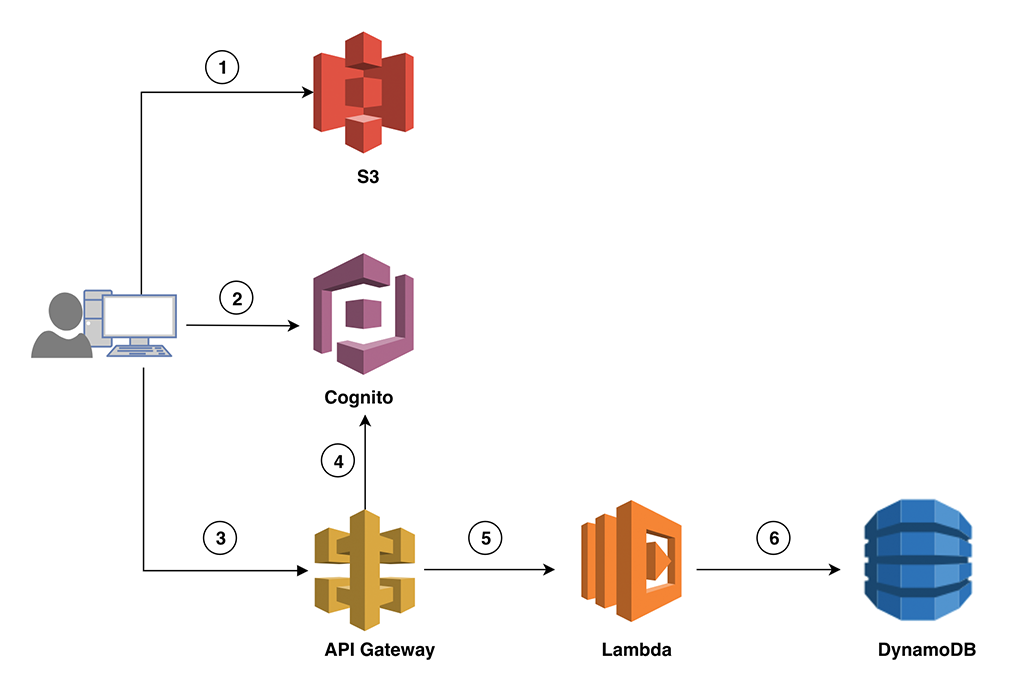
# Sécurité :



Ci-dessus le fonctionnement de la sécurisation de l’api.

Chaque utilisateur est identifié lorsqu’il se connecte et effectue les requêtes avec le token qui lui est renvoyé en paramètre.

Il est à inclure dans le header, avec l’attribut **Authorization**

Ainsi il est possible de différencier si un utilisateur est un administrateur ou non afin de valider ces actions.

Le schéma ci-dessus présente le fonctionnement d’application :

* 1. l’utilisateur requête l’application qui est stockée sur S3
* 2. Il passe ses identifiants à Cognito afin qu’un token lui soit renvoyé
* 3. Il effectue des appels à l’API avec le token qui lui à été donné
* 4. l’API exécute le code présent dans la lambda qui lui est attribuée
* 5. la lambda récupéré les informations depuis DynamoDB qui est une base noSQl