Institution des Chartreux

Rapport de stage

GameVerse

ADAM Emilien
03/03/2023

Remerciements

Je voudrais d'abord remercier Madame Couder, pour avoir relayé cette offre de stage qui m'a permis d'intégrer l'entreprise.

Je remercie également Ugo MIGNON pour m'avoir accepté au sein de l'entreprise, mais également pour le support et l'assistance qu'il m'a apporté durant tout le stage.

Je tiens également à remercier Jordan LABROSSE pour le suivi tout le long du stage et pour les nombreuses aides apportées.

Je remercie également Monsieur Lourenco pour s'être rendu disponible afin de venir visiter et de s'être intéressé au projet.

Enfin je voudrais remercier COTET Landry qui m'a permis d'intégrer l'entreprise pour réaliser ce stage.

Introduction

Je suis en deuxième année de BTS Service Informatique aux Organisations (SIO), en fin de cycle préparatoire a une formation d'ingénieur en cybersécurité. Afin de valider mon année ainsi que de gagner en expérience professionnelle. Je suis actuellement intéressé par de nombreux domaines dans l'informatique, l'analyse de données et le Big Data, le Dev Ops ou encore le pentesting. Entrant dans mon cycle d'ingénieur l'année prochaine, je souhaite en apprendre plus sur ces différents domaines et de pouvoir les approfondir, tout en restant dans mon domaine de spécialité, le développement d'applications. Je réalise donc un stage dans l'entreprise GameVerse afin de monter en compétences techniques dans le développement d'applications.

Contexte

GameVerse est une start-up née en 2021 par des entrepreneurs Français. Créée par des anciens élèves de CPE, formation ICS, GameVerse a pour but de révolutionner les relations des joueurs de jeux-vidéos.

GameVerse est une application en développement actif, qui vise à centraliser toutes les différentes plateformes de jeu mais surtout qui vise à faciliter la rechercher de co-équipiers de qualité. Le projet principal de GameVerse est d'utiliser une intelligence artificielle couplée à un système de matchmaking pour que les utilisateurs trouvent le ou les co-équipiers les plus adaptés. A terme, GameVerse sera disponible sur PC, mobile, Console ou toute plateforme. GameVerse centralisera donc les différentes plateformes et Game providers (Steam, Origin, Epic Games…) pour servir de lanceur de jeu, mais également pour afficher les statistiques du joueur, lui proposer d'autre contenu.

Le but final de GameVerse sera d'axer la plateforme plus sur un aspect eSport et compétitif, avec des options de création d'équipes, de suivi ou de coaching. La plateforme sera une solution pour améliorer ses performances.

GameVerse est une application gratuite, qui offrira des fonctionnalités supplémentaires contre un abonnement mensuel sous la forme d'un modèle donc Freemium.

Projet

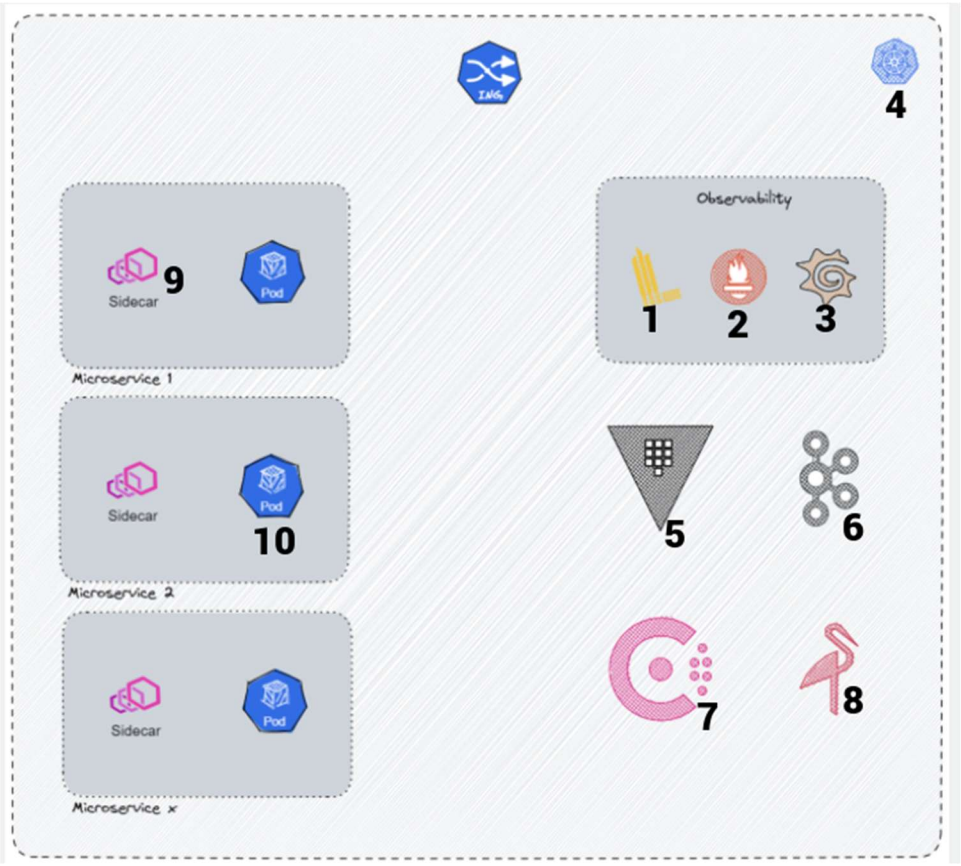
L'application GameVerse est disponible sur de nombreuses plateformes, Navigateur web, Mobile, Windows, MacOs et compte être accessible depuis n'importe quelle plateforme possible. L'application recevra donc des connexions depuis de nombreuses sources, que ce soit depuis l'application mobile, depuis un navigateur ou dans le futur donc, depuis une console par exemple. Avoir une connexion sécurisée à son application est aujourd'hui un élément extrêmement important mais également complexe, d'avantage lorsque l'application fonctionne avec de nombreux micro-services et reçoit des connexions depuis de nombreuses sources différentes. Pour cela GameVerse utilise donc une solution gratuite et open source Ory Network. Ory Network permet d'héberger localement une infrastructure complète te gestion d'accès, de fournisseur d'identité et de server oAuth2.0. Ory est le concurrent de la solution la plus répandue actuellement, Keycloack, également open source.

Ma tâche sera donc d'assister à l'implémentation de cette solution d'Identity Access Management (IAM) ainsi que de gérer les connexions depuis les appareils mobiles.

L'infrastructure de GameVerse est découpée en micro-services hébergées et gérées par eux-mêmes. Le choix de tout découpé en micro-services est pour assurer une bonne scalabilité de l'infrastructure et assurer un déploiement presque instantané de tous les services si besoin. Pour cela GameVerse utilise Kubernetes, une solution open source pour gérer les workloads (charges de travail) et permet de conteneuriser les services. Kubernetes permet d'adapter les ressources déployées (Server d'API, Base

de données...) pour l'infrastructure grâce à un load balancing. Pour ce qui est de la scalabilité des bases de données, GameVerse utilise CockroachDB.

Schéma de l'infrastructure :



1. Grafana Loki
2. Prometheus
3. Grafana
4. Kubernetes
5. Vault
6. Kafka
7. Consul
8. Minio
9. Envoy
10. Micro-service

Dans le cadre de ce stage, il a été imposé d'utiliser Podman lors de l'utilisation de conteneurs. Podman fonctionne comme Docker, mais ne nécessite pas forcément des droits d'administrateurs ou root. De plus, podman est plus adapté pour créer des pods Kubernetes, d'où son utilisation dans GameVerse. De plus, l'application GameVerse est écrit en TypeScript avec le framework NuxtJS.

Déroulé

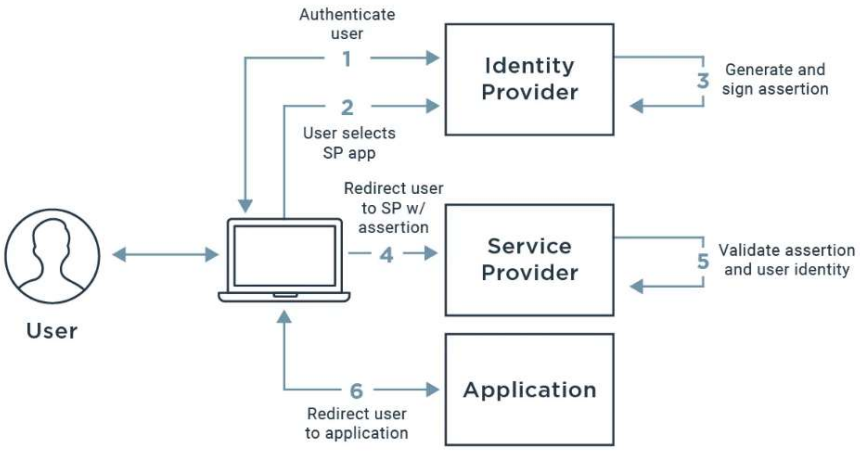
Le début du projet a d'abord été l'apprentissage des normes actuelles des différents protocoles utilisés pour gérer les accès et fournir des identités.

La méthode traditionnelle historique pour gérer les connexions est un formulaire dans lequel l'utilisateur rentre ses données. Ces dernières sont ensuite envoyées vers le server puis, si la connexion est valide, l'utilisateur est stocké en session. Bien que cette méthode puisse être sécurisée par des protocoles de communications comme le HTTPS, cette méthode est aujourd'hui obsolète et remplacée par des nouveaux protocoles et flow de communication. Elle est également inadaptée pour des applications de larges envergures ou des systèmes avec des connexions entrant de plusieurs plateformes ou clients différents.

Security Assertion Markup Langage

Le Security Assertion Markup Langage (SAML) est un protocole d'échange de messages standardisé utilisé pour l'authentification et l'autorisation entre différents systèmes. Il permet à un utilisateur de se connecter une seule fois à un Identity Provider (fournisseur d'identité) et de bénéficier ensuite d'un accès transparent et sécurisé à plusieurs services.

Lorsqu'un utilisateur tente d'accéder à un service protégé, le service redirige l'utilisateur vers l'Identity Provider. L'utilisateur doit alors s'authentifier auprès de l'Identity Provider, qui lui transmet ensuite un jeton d'assertion SAML contenant des informations sur l'identité et les autorisations de l'utilisateur. Ce jeton est signé numériquement pour garantir son intégrité et sa validité.



Le protocole SAML utilise des fichiers XML afin de définir comment gérer les utilisateurs. Ces fichiers de configuration XML jouent un rôle essentiel dans la mise en place de la communication entre les différentes entités d'un échange SAML. Ces fichiers permettent de spécifier les différents paramètres nécessaires à l'authentification et à l'autorisation des utilisateurs, ainsi que les informations de configuration nécessaires pour chaque entité impliquée dans l'échange. Le protocole SAML permet donc une implémentation du système de Single Sign On (SSO) qui permet donc à un utilisateur de se connecter qu'une fois à un service pour ensuite disposer d'accès à d'autres services partenaires.

JSON Web Token

Les tokens utilisés pour l'échange d'informations d'authentification et d'autorisation dans les protocoles modernes sont souvent basés sur la norme JSON Web Token (JWT). Cette norme de token respecte les règles de confidentialité, d'intégrité et garantit l'identité des deux entités communicantes.

Les avantages de la norme JWT sont nombreux. Elle utilise une clé privée / publique sous forme d'un certificat pour garantir la sécurité de l'échange. La signature par JSON est simple et la plupart des langages de programmation peuvent traiter le format JSON. De plus, le JWT est utilisé à l'échelle d'Internet, ce qui facilite la gestion des JSON Web Tokens côté client sur plusieurs plateformes, notamment mobiles.

Un token JWT est généralement séparé en trois parties : le Header, le Payload et la Signature. Ces trois parties sont ensuite encodées (Base64) de manière à obtenir un token sous la forme xxxx.yyyy.zzzz.

La partie Header est composée du type de token (JWT) et de l'algorithme de signature utilisé. La partie Payload contient des données nécessaires à la demande, telles que les informations d'identification de l'utilisateur ou les autorisations d'accès à certains services.

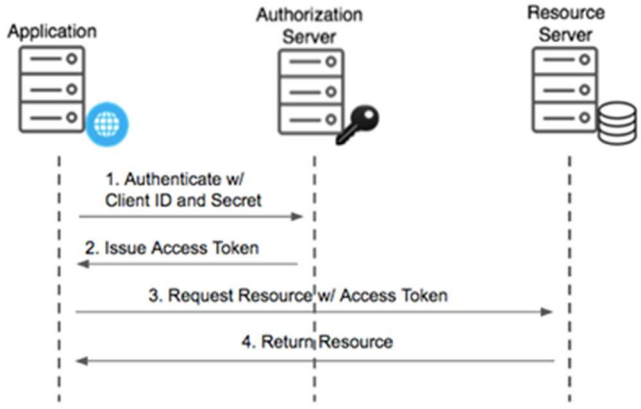
La partie Signature permet de garantir l'intégrité du token en vérifiant que celui-ci n'a pas été altéré pendant son transit. Cette partie est générée à partir des deux parties précédentes (Header et Payload) et d'une clé de signature, qui peut être une clé privée / publique.

OAuth 2.0

Norme de l'industrie pour les protocoles d'autorisation. C'est un protocole d'autorisation et non d'authentification. Permets à une application d'accéder à des ressources au nom d'un utilisateur. Elle fournit un accès et limite les actions que l'application cliente peut réaliser sur les ressources au nom de l'utilisateur sans jamais partager les informations d'identification de l'utilisateur. Utilise des jetons d'accès qui représentent l'autorisation d'accès. Ne définit pas de format spécifique pour les jetons d'accès.

Rôles de la norme :

- Propriétaire des ressources
- Client : demandeur des ressources, doit détenir le jeton
- Server d'autorisation : reçoit les demandes de jetons, et les délivre après autorisation
- Server de ressources : protège les ressources, reçoit les demandes d'accès des clients et renvoie les données appropriées Pour une meilleure sécurité, les servers d'autorisation peuvent retourner un code d'autorisation qui est ensuite échangé contre un jeton d'accès

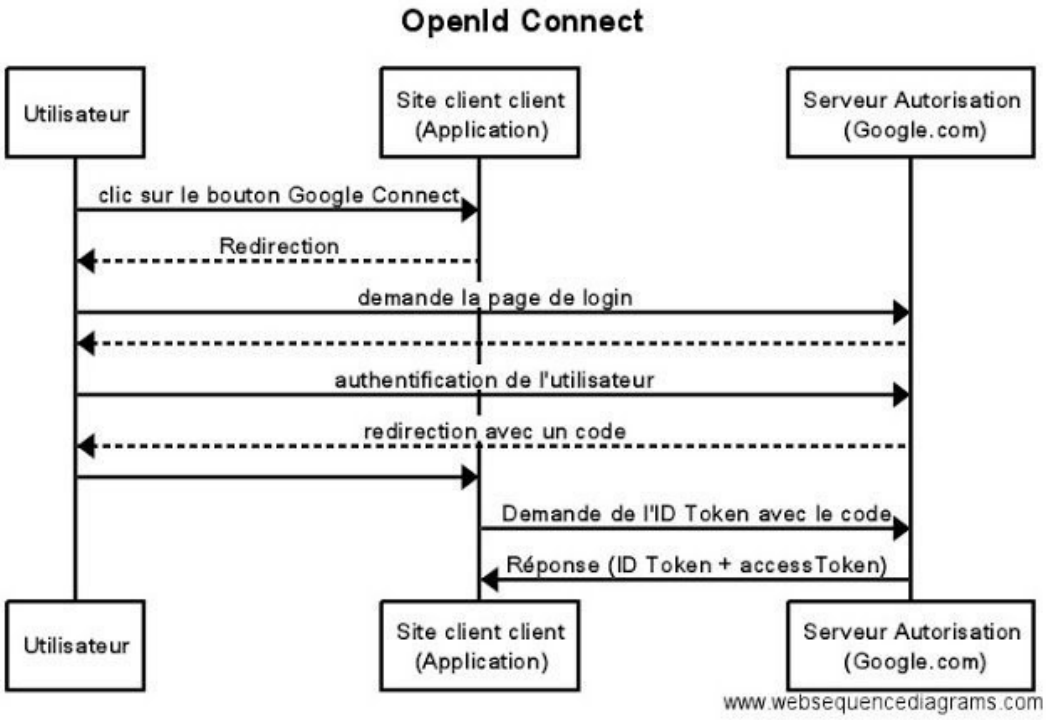


OIDC

OIDC est un protocole d'authentification et d'autorisation qui se base sur OAuth 2.0. Il permet d'ajouter une couche d'identité à OAuth 2.0. Au lieu de simplement fournir un jeton d'accès pour accéder à des ressources, OIDC fournit également des informations sur l'identité de l'utilisateur, telles que son nom et son adresse e-mail.

OIDC permet également une meilleure gestion des sessions d'utilisateur, ce qui permet de mettre en place le Single Sign-On (SSO) sur plusieurs applications. Par exemple, lorsqu'un utilisateur se connecte à une application avec son compte OIDC, il peut ensuite accéder à d'autres applications sans avoir besoin de se connecter à chaque fois.

En termes de sécurité, OIDC permet également de mettre en place des flux d'authentification et d'autorisation plus robustes, en utilisant des mécanismes tels que la vérification de l'adresse e-mail ou le consentement explicite de l'utilisateur pour accéder à certaines ressources.



Keycloack

Keycloack était la solution choisie par GameVerse pour gérer l'authentification des utilisateurs. Keycloack est une solution d'IAM open source développée par Red Hat. Keycloack permet d'intégrer à son interface et son application une gestion sécurisée des authentifications. Cependant, cette interface n'était pas totalement personnalisable et n'est donc pas adaptable à GameVerse. De plus, Keycloack n'est pas découplable en différents micro-services, qui n'est donc pas scalable ou adapté à des situations de pannes.

Ory Network

Ory est le concurrent émergeant de Keycloack. C'est une solution open source d'infrastructure d'authentification qui peut être hébergée localement. Cette solution est notamment utilisée par OVH. Le but d'utiliser Ory est de déléguer la partie de la sécurisation des authentifications, partie très technique et complexe pour obtenir un bon résultat. Ory Network est découpé en plusieurs sous éléments qui peuvent chacun être utilisés à part et ont chacun leur rôle à jouer dans la gestion

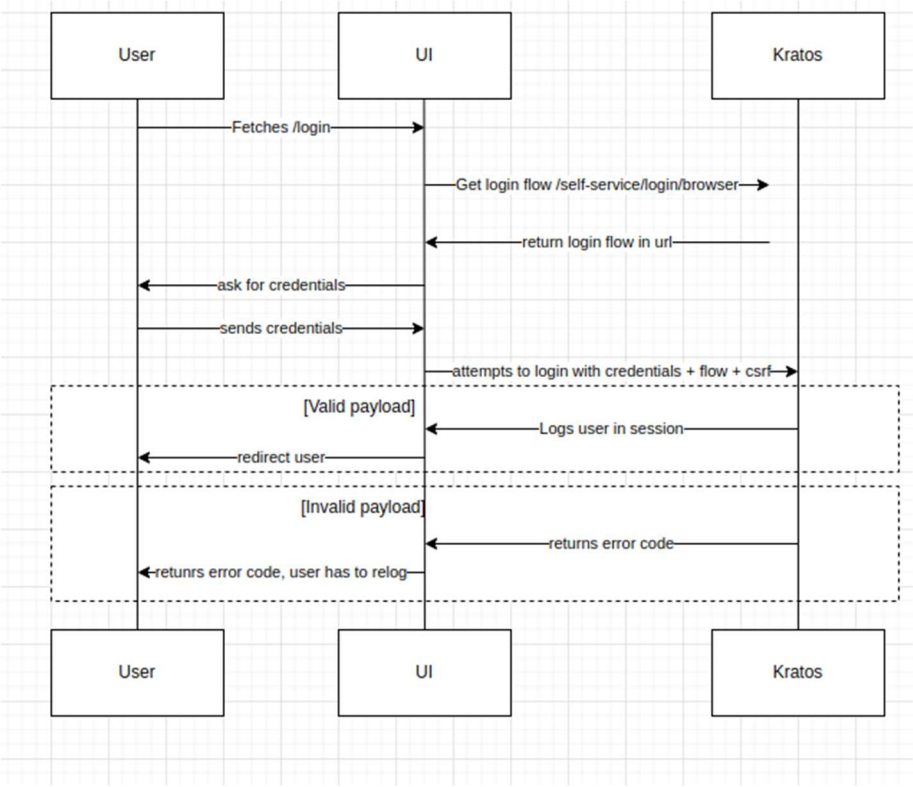
d'identité, d'authentification et d'accès aux informations. Ces différentes parties sont toutes découpées en micro-services :

- Ory Kratos
- Ory Hydra
- Ory Keto
- Ory oathkeeper

Ory permet donc une gestion des accès et une gestion de l'authentification scalable et adapté face aux situations de pannes.

Ory Kratos

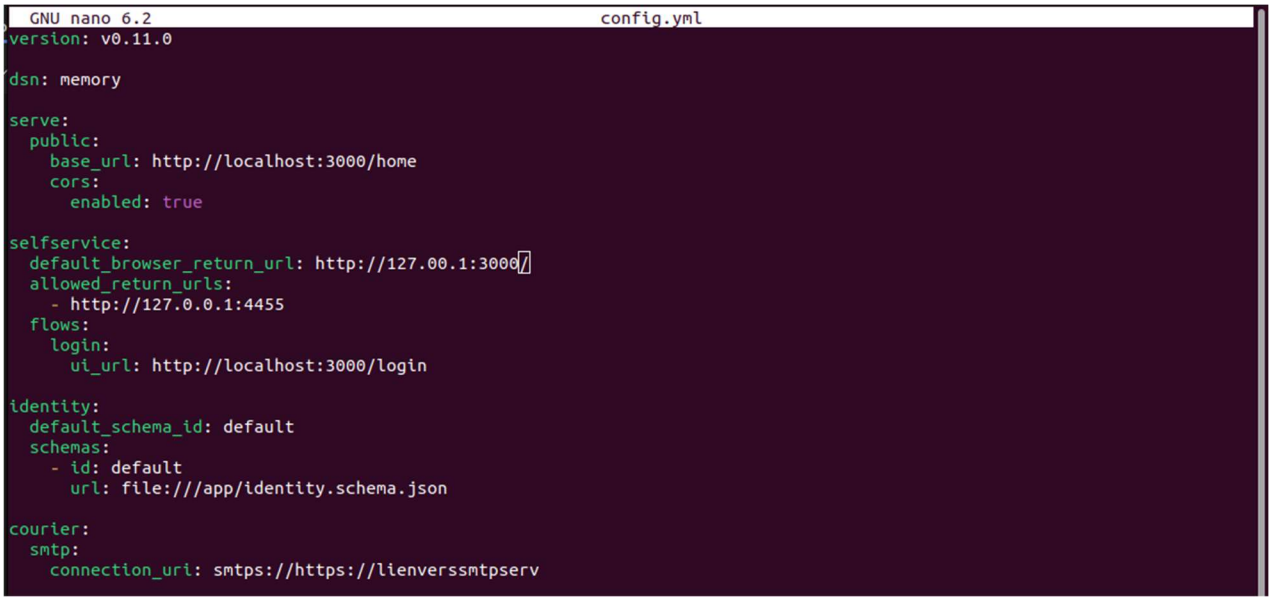
Ory Kratos est un système de management d'identité et d'utilisateurs par API. Ory Kratos permet de gérer la majorité des tâches que les applications modernes doivent gérer comme la création d'identité et l'authentification, le 2FA/MFA, la vérification de compte par mail... Ory Kratos permet donc d'implémenter facilement une solution sécurisée pour gérer les connexions et authentification de son application. Pour que Kratos valide une authentification, il faut fournir un login flow retourné par Kratos même. Ce flow est obtenu en appelant un end point de kratos et est retourné dans l'url. Kratos gère également les attaques de Cross-Site Request Forgery (CSRF).



Exemple de flow obtenu après avoir rentré http://kratos/self-service/login/browser:

Pour faire fonctionner Kratos, Ory met à disposition différentes conteneurs Docker modifiable facilement à l'aide de simples fichiers de configuration afin de l'adapter à ses besoins.

Voici par exemple mon fichier de configuration pour exécuter Kratos:



La “base_url” correspond à l’url de base de l’application vers laquelle Kratos redirige les utilisateurs.

Pour ce qui est du self-service, c’est l’Endpoint de l’API de Kratos qui gèrera les authentifications des utilisateurs. Ici, une fois le login flow complété (donc que l’authentification est valide) retourne vers le /login du server web (ici hébergée sur le port 3000). La partie Identity elle concerne les informations d’un utilisateur et le schéma qu’elles devront suivre, j’ai ici utilisé le schéma basique fournis par Ory.

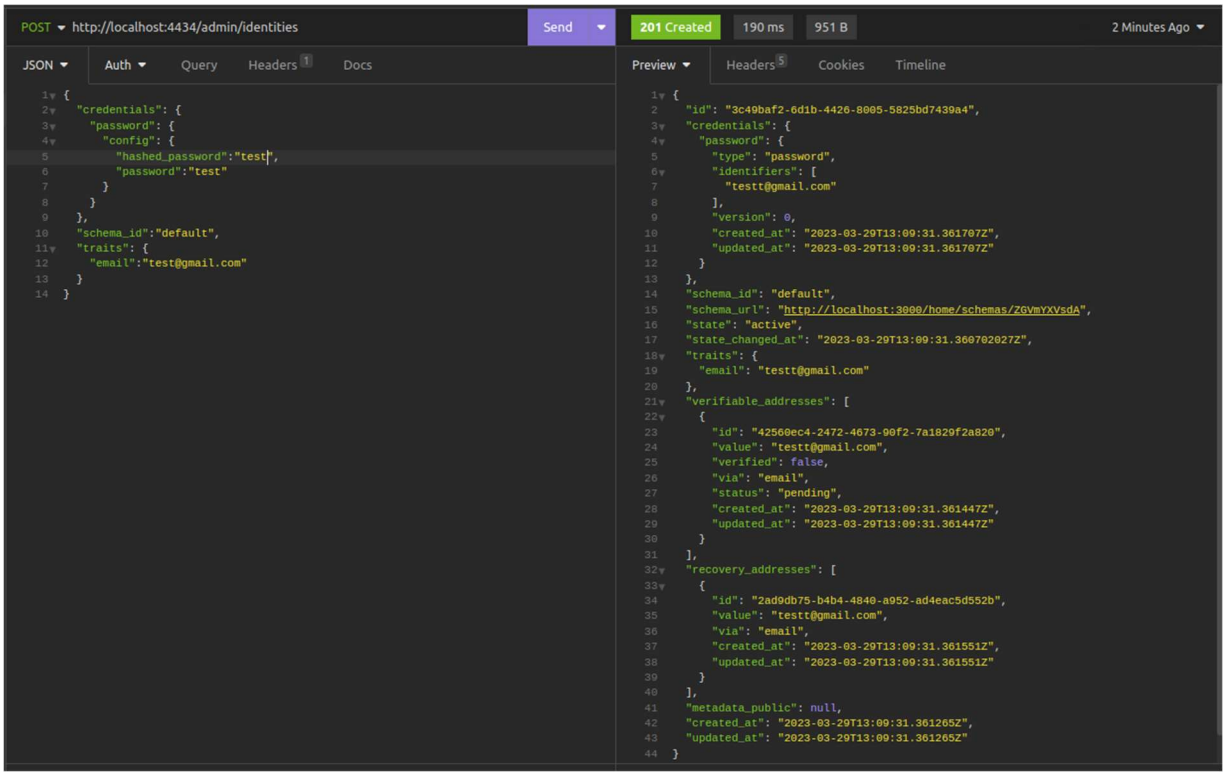
Pour ce qui est de la partie courrier, elle désigne le server smtp qui sera utilisé pour les vérifications des adresses emails des utilisateurs, ainsi que pour les changements de mots de passes. Pour mon cas, n’étant que dans des environnements de tests, j’ai simplement remplacé l’url par n’importe quel domaine HTTPS.

Voici ensuite la commande podman que j’ai utilisé pour lancer Kratos, qui utilise des volumes afin de transférer des fichiers dans le conteneur :

$ podman run -it --name kratos --mount type=bind,source=config.yml,target=/app/config.yml --mount,type=bind,source=identity.schema.json,target=/app/identity.schema.json -e LOG_LEAK_SENSITIVE_VALUES=true -e DSN="memory" -p 4433:4433 –p 4434:4434 kratos serve --config /app/config.yml

Kratos utilise deux ports, le 4433 et 4434, un pour l’interface administrateur et un pour le self-service. La variable d’environnement est ici attribué à memory. DSN désigne le type de stockage que va utiliser Kratos pour stocker les identités ect (SQLlite, POSTGRES...). En lui attribuant la valeur memory, alors les données sont gardées en mémoire de manière temporaire et sont supprimées à chaque redémarrage de Kratos. La variable LOG_LEAK_SENSITIVE_VALUE est attribuée à true afin d’avoir le retour des erreurs complètes dans la console de Kratos et de pouvoir donc mieux corriger les erreurs lors des requêtes. Je n’ai qu'utilisé Kratos de manière locale et pour des tests, ces memory et true sont donc des paramètres acceptables tant que l’on reste dans un environnement de production.

Ensuite, pour créer une identité, il suffit d’envoyer une requête POST vers l’endpoint /admin/identities avec en payload JSON les données de l’utilisateur, l’API renvoie ensuite l’utilisateur avec ses informations complètes.



Ory Hydra

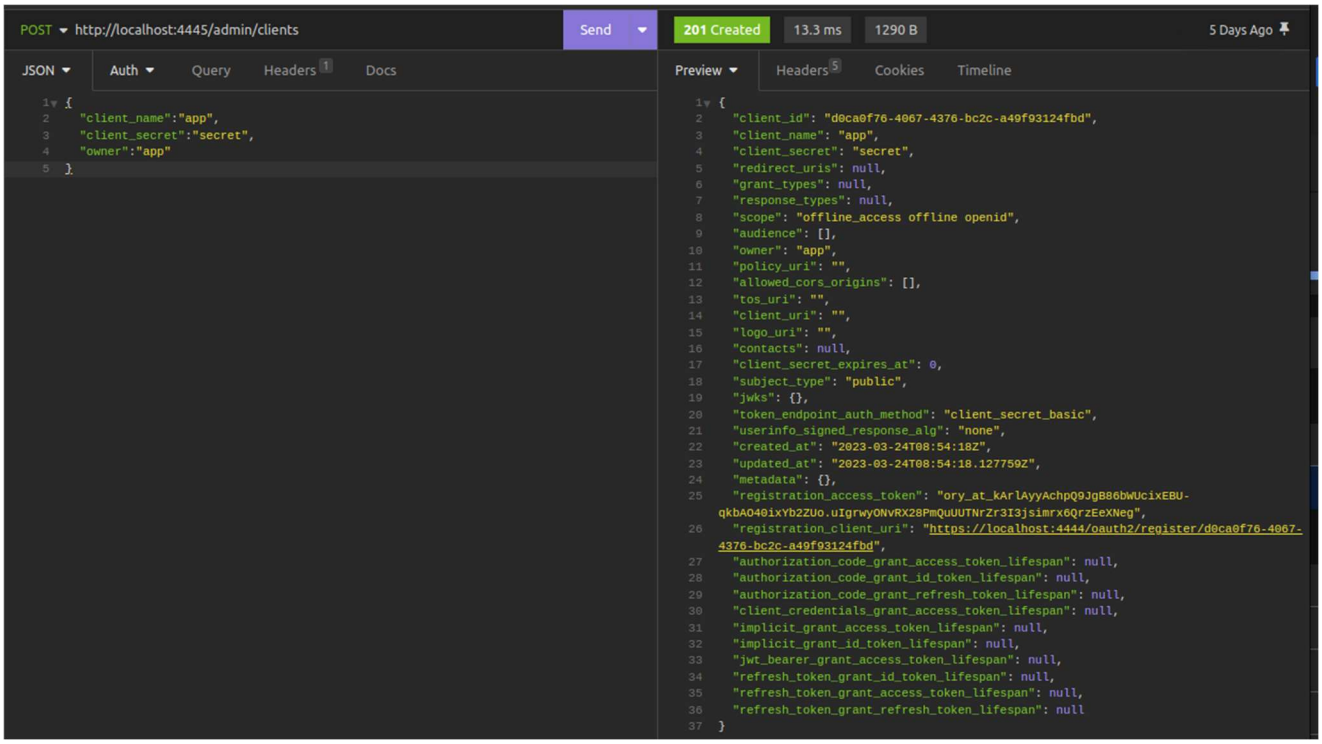
Ory Hydra sert de provider OIDC et OAuth 2.0. Hydra permet donc la connexion depuis différents appareils et de garder ces appareils en mémoire, mais également la création d'applications complète nommées clients. Hydra permet également de créer des tokens de connexion pour les différentes devices ou client et de créer des dates d'expiration pour ces tokens. Les tokens utilisés par Hydra suivent la norme JWT. Pour exécuter Hydra, de même qui pour Kratos Ory met à disposition différents conteneurs Docker configurable par un fichier de configuration.

Pour exécuter Hydra j'ai utilisé le conteneur de base :

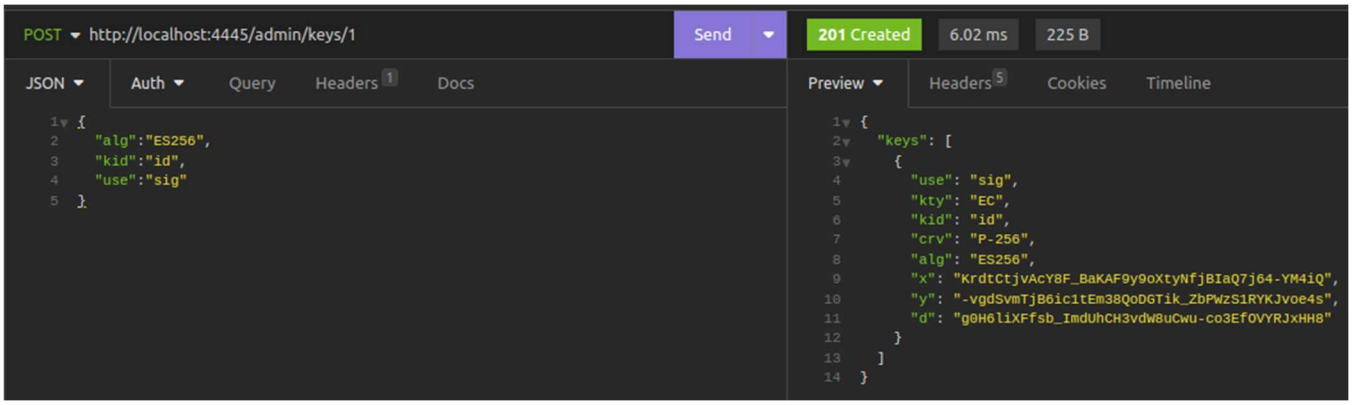
$ podman run -it -e SECRETS_SYSTEM="secret" -e LOG_LEAK_SENSITIVE_VALUES=true -e DSN=memory -p 4445:4445 –p 4444:4444 docker.io/oryd/hydra

De même que pour Kratos, Hydra nécessite deux ports, un pour la partie administrateur et un pour le self-service. Pour ce qui est des deux variable d'environnements LOG_LEAK_SENSITIVE_VALUES et DSN, elles sont attribuées à true et memory pour les mêmes raisons que Kratos.

Une fois que Hydra fonctionne, pour créer un client, il suffit de fetch l'endpoint /admin/clients/ et inclure dans le payload les informations du client et Hydra retourne les informations du client.



Hydra permet également donc de créer des sets de clés. Pour cela il faut utiliser l'endpoint /admin/keys/{id} avec une requête POST. L'id est à définir soir même dans l'url et il faut inclure un payload avec les informations de la clé qui sont l'algorithme de hachage, l'id de clé et l'utilisation de la clé (signature / encrypt). On retrouve alors bien le schéma décris précédemment pour les clés de la norme JWT.



Ory Keto

Keto est la partie de Ory qui permet la gestion de permissions d'accès aux ressources par les utilisateurs. Comme pour Hydra et Kratos, Keto est utilisable via des conteneurs Docker et est donc également paramétrable par des simples fichiers de configuration.
Keto fonctionne avec des namespaces, qui représentent une certaine partie des permissions. Les namespaces sont faits de règles et d'objets. Ces derniers peuvent être permis ou non d'interagir entre eux. Pour commencer à utiliser Keto il faut donc d'abord paramétrer les namespaces dans le fichier de configuration de la manière suivante :

namespaces:

- id : int

name: nom_namespace

Voici un exemple du fichier de configuration pour la gestion d'accès à des fichiers :

namespaces:

- id: 0

  name: Group

- id: 1

  name: Folder

- id: 2

  name: File

- id: 3

  name: User

Ensuite, pour ce qui est de la création de relations et en vérifier, comme pour Kratos et Hydra, Keto fonctionne par une API. Pour créer une relation il suffit d'envoyer une requête PUT à l'endpoint /admin/relation-tuples (port 4467 admin par défaut) et rajouter un payload json avec les informations de la relation. Voici le format :

```
{
    "namespace": "namespace_name",
    "object": "object_name",
    "relation": "relation_name",
    "subject_set": {
    "namespace": "second_namespace_name",
    "object": "second_object_name     ",
    "relation": "relation_name"
    }
}
```

Pour vérifier une permission il faut envoyer une requête GET à /relation-tuples/check et passer les informations suivantes dans l'url :

- Namspace
- Object

- Relation
- Subject_id
- Subject_set.namespace
- Subject_set.object
- Subject.set_relation (obligatoire, ou mettre à vide)

En reprenant l'exemple précédent, voici un exemple de requête

/relation-tuples/check?namespace=Folder&object=keto%2F&subject_set.namespace=Group&subject_set.object=developer&subject_set.relation=members&relation=viewers

Keto retourne alors sous forme de JSON true ou false selon la relation:

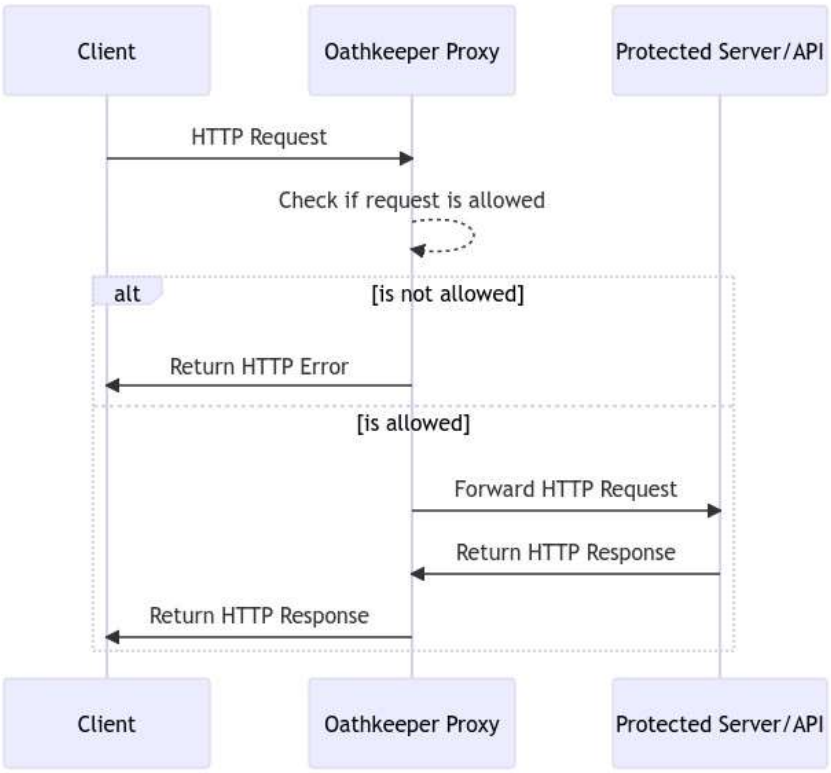
{

"allowed":"true/false"

}

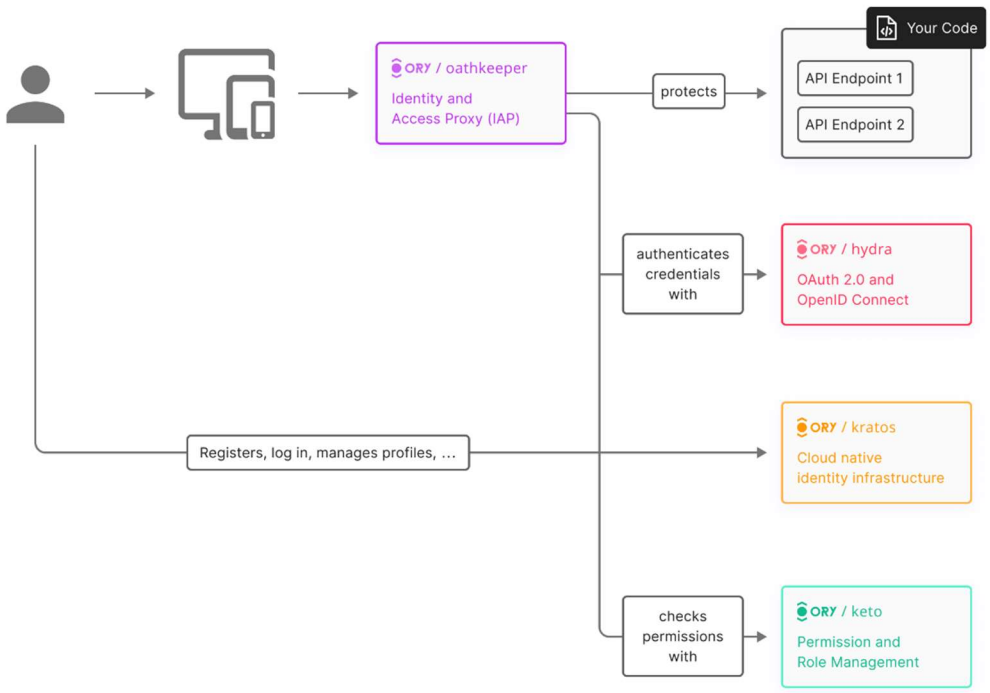
Ory oAthkeeper

Ory oAthkeeper peut avoir plusieurs fonctions différentes. Son but est de filtrer les requêtes HTTP entrante et de les autoriser ou non. OAthkeeper peut donc être utilisé autant come un Reverse Proxy que comme API de prise de décisions. Dans son implémentation de GameVerse, oAthkeeper est utilisé comme un reverse proxy.

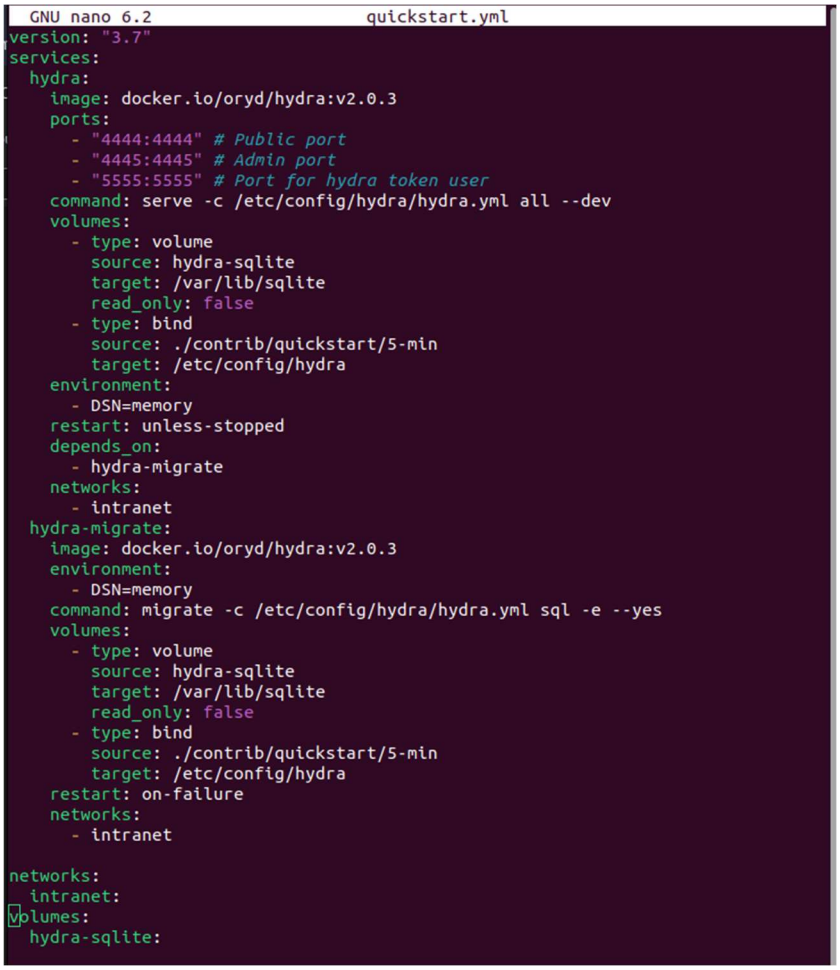


Flow complet

Ces éléments de Ory peuvent donc fonctionner et être déployés seul. Cependant ils peuvent également être tous combinés afin d'obtenir une infrastructure et gestion d'identité et d'authentification complète. Voici un schéma montrant le flow et le rôle de chaque service d'Ory lorsque tous les éléments sont liés.



Dans le cadre de mon stage et GameVerse ayant déjà un Reverse Proxy, je n'ai que relié les services Kratos et Hydra, pour gérer les connexions depuis les appareils mobiles. Pour connecter Kratos et Hydra et utiliser Hydra comme oAuth provider, il faut tout d'abord générer dans Hydra un Access Token pour valider Hydra comme application cliente. Pour cela j'ai utilisé le fichier de configuration de Hydra suivant (fourni par Ory) :



Puis j'ai lancé le contenu Docker avec la commande suivante :

$ podman-compose -f quickstart.yml up --build

Il faut maintenant créer le client :

$ client=$(podman-compose -f quickstart.yml exec hydra \

hydra create client \

--endpoint http://127.0.0.1:4445/ \

--format json \

--grant-type client_credentials)

Puis récupérer l'ID et le secret de ce client :

$ client_id=<id>

$ client_secret=<secret>

Et enfin récupérer le token crée par Hydra :

$ docker-compose -f quickstart.yml exec hydra \

hydra perform client-credentials \

--endpoint http://127.0.0.1:4444/ \

--client-id $client_id \

--client-secret $client_secret

Il faut désormais lire Hydra à Kratos dans le fichier de configuration de Kratos en rajoutant cet élément à la fin du fichier :

oauth2_provider:
  headers:
Authorization: Bearer <access_token>
  url: http://127.0.0.1:4455

Désormais, lors de d'authentifications de l'utilisateur, Kratos transmettra automatiquement à Hydra un "login consent" qu'Hydra utilisera pour soit afficher l'utilisateurs les interfaces d'autorisation d'accès pour l'appareil à la connexion, soit validé la bonne connexion si les tokens associés sont valides.

Le lien entre Kratos et Hydra est nécessaire pour l'application mobile, afin de différencier les connexions mobiles des connexions browser ou de l'application de bureau. Pour cela, j'ai donc également expérimenté avec l'application mobile de GameVerse afin de gérer les connexions sur appareil mobile.

NuxtJS

Nuxt est le framework open source JavaScript Français utilisé par GameVerse pour leur application. Nuxt permet la création d'une arborescence complète avec une simple commande et force à séparer le code dans l'architecture MVC. Nuxt est compatible avec de nombreuses bibliothèques JavaScript et permet également de les importer automatiquement. GameVerse a également choisi NuxtJS pour sa rapidité d'exécution mais aussi par la grande communauté derrière le projet open source, qui travaille constamment dessus afin d'améliorer le framework. De plus, les développeurs de NuxtJS travaillent en coopération avec ceux de VueJS afin d'offrir la meilleure compatibilité entre les deux technologies.

La particularité de NuxtJS que j'ai le plus utilisé est les Stores. Les stores permettent de gérer l'état global de l'application. Un store est une instance de l'application qui permet de stocker diverses données. Les stores peuvent ensuite être appelés et utilisé partout dans l'application afin d'altérer ou récupérer leurs données. Les stores peuvent également réaliser des actions comme des appels d'API selon certaines conditions réunies ou selon certains appels. Cette possibilité d'appel est utile pour GameVerse afin de pouvoir chercher les informations de connexion auprès de Kratos si l'utilisateur n'est pas authentifié lors du lancement de l'application ou lors du chargement d'une page. Un avantage de ces stores est également le fais qu'ils peuvent être stocké côté server, mais également côté client ou bien des deux côtés.

Un autre composant de Nuxt est les middlewares. Ces derniers permettent d'exécuter du code avant ou après le chargement d'une page Vue. Le middleware peut effectuer des tâches comme de la validation de données, la gestion d'erreurs ou encore gérer l'authentification. Un middleware peut être global, c'est-à-dire qu'il prendra action sur chaque vue différente, ou alors il peut être attribué à une unique ou plusieurs vues manuellement. Pour définir un middleware global, il suffit d'ajouter .global au nom du fichier du middleware. Pour affecter manuellement un middleware à une vue, il suffit de l'importer dans le code de la vue.

```
export default {
  middleware: [nom_middleware],
}
```

Les middlewares m'ont été utile pour gérer si les utilisateurs sont authentifiés au non, grâce à une middleware global qui lors du chargement d'une page vérifié le store qui gère les utilisateurs. Si ce dernier est vide, alors l'utilisateur doit se connecter et tout le flow doit être initié avec une redirection vers la page de connexion.

src > middleware > TS auth.global.ts > ...

```ts
import { useAuthStore } from '~/stores/auth'

const excludePages = ['/login', '/register', '/forgot']
export default defineNuxtRouteMiddleware(async (to, from) => {
  if (process.client) {
    const authStore = useAuthStore()

    // If the user is not authenticated, redirect to login page
    if (!authStore.isAuthenticated) {
      if (!excludePages.includes(to.path)) {
        return navigateTo('/login')
      }
    } else {
      if (excludePages.includes(from.path) && to.path !== '/home') {
        return navigateTo('/home')
      }
    }

    // If the user is authenticated, continue to the page
    return
  }
})

```

CapacitorJS

Pour son application Mobile (Android / iOS) GameVerse utilise CapacitorJS, qui permet de directement générer une application Android / iOS depuis le code NuxtJS en TypeScript. C'est donc avec Capacitor qu'il faut gérer le flow de connexion depuis les appareils mobiles. Pour se faire, il a été choisi d'ouvrir un navigateur web dans l'application même qui va donc lui-même charger la page web de login de GameVerse pour ensuite initier le flow de connexion normal. Pour ouvrir le navigateur au sein même de l'application mobile, Capacitor dispose d'un plugin browser, qui permet d'ouvrir un navigateur directement sur la page voulue. Dans le code suivant il y a également un test afin de s'assurer que l'utilisateur soit bien sur un appareil mobile grâce à .isNativePlatform() qui retourne un booléen selon si l'utilisateur est ou n'est pas sur un appareil mobile.

```
11    if (!authStore.isAuthenticated) {
12      //If user is on mobile, open browser
13      if (Capacitor.isNativePlatform()) {
14        const openCapacitorSite = async () => {
15            await Browser.open({ url: 'https://gameverse.app/login' });
16        };
17
18        await openCapacitorSite();
19      }
```

Liens Profonds Android (Deep-links)

Ma dernière tâche a été de créer un lien profond pour l'application Android de GameVerse. Un lien profond signifie un lien, qui lorsqu'il est cliqué ouvre directement l'application mobile sur le téléphone si la personne en utilise un. Les deep-links peuvent permettre également de rediriger vers une page spécifique de l'application mais également de transférer des informations par l'URL. Je n'ai pas pu générer de deep-link pour iOS n'ayant pas accès à XCode disponible seulement sur Mac. Pour générer un lien profond Android, il faut d'abord héberger un fichier JSON sur une adresse web à la destination suivante : https://domain.name/.well-known/assetlinks.json. Ce fichier JSON est le résultat de la signature de l'application en SHA256. Voici le fichier de GameVerse:

```
[
  {
    "relation": [
      "delegate_permission/common.handle_all_urls"
    ],
    "target": {
      "namespace": "android_app",
      "package_name": "app.gameverse",
      "sha256_cert_fingerprints": [
        "1E:9D:9F:11:32:EB:C0:7F:DB:FA:75:71:95:F9:DE:57:03:CE:B0:EB:77:B0:6F:4B:A5:31:96:24:F0:1D:00:60",
        "E3:DE:74:6B:13:93:CA:A9:98:0D:37:B4:63:10:2A:5D:DC:00:A1:04:81:91:02:EF:EB:42:91:CB:BB:36:2B:CC"
      ]
    }
  }
]
```

Ensuite, dans l'application Android, il faut modifier le fichier AndroidManifest.xml en ajoutant le code suivant, et l'adapter selon ses besoins.

```
<intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
</intent-filter>

<intent-filter android:autoVerify="true">
    <action android:name="android.intent.action.VIEW" />

    <category android:name="android.intent.category.DEFAULT" />
    <category android:name="android.intent.category.BROWSABLE" />

    <data
      android:scheme="https"
      />


    <data
        android:host="gameverse.app"
        android:pathPrefix="/deep-links" />
</intent-filter>
```

Pour le cas de GameVerse donc, https://gameverse.app/deep-links ouvrira l'application mobile si cette dernière est installée sur le téléphone mobile.

Conclusion

Dès le début de ce stage j'ai été en autonomie, pour l'apprentissage des différentes anciennes et actuelles normes de connexion et d'authentifications. Mes principaux problèmes lors de ce stage ont été principalement lors de l'exécution des différentes images et conteneurs Docker sur les bon paramètres des fichiers de configuration. La principale difficulté a notamment été sur la liaison entre Ory Kratos et Ory Hydra, où malgré les deux services liés via le fichier de configuration. Ory Hydra recevait bien des éléments dans sa console mais le login_consent n'était pas reçu correctement.

Ce stage m'a permis d'apprendre de façon autonome sur des technologies récente en les comparant aux anciennes. J'ai pu découvrir les protocoles d'authentification que sont l'OIDC et l'OAuth 2.0 et comprendre et découvrir comment ils sont aujourd'hui utilisés pour gérer les authentifications et les connexions des utilisateurs.

J'ai appris comment certaines infrastructures sont découpées en différents micro-services qui peuvent ensuite communiquer ensemble ou de leur côté. J'ai approfondi ma maitrise de Podman ainsi que des différents éléments de Docker. J'ai découvert un nouveau Framework JavaScript et pratiqué le language TypeScript. J'ai également découvert CapacitorJS

Sur les servers de productions il reste le lien à réaliser le lien entre Ory Kratos et Ory Hydra. Ory oAthkeeper est bien fonctionnel et fonctionnel déjà avec les différents web sockets des micro-services. Pour ce qui est du front de l'application, l'authentification auprès de Kratos est bien fonctionnelle, le login flow et tous les autres procédés comme le MFA, vérification de mail ect fonctionnent correctement. Pour ce qui est de l'application mobile, pour gérer le flow d'authentification avec les autorisations nécessaires lorsque l'utilisateur est sur appareil mobile.