Introduction à nextAuth

Je tiens à préciser que les informations et les configurations que j'ai partagées ici sont tirées directement d'une ressource éducative en ligne : la vidéo YouTube produite par Melvynxdev. Ce contenu est reproduit ici avec l'intention de servir de support pédagogique et de renforcer la compréhension des processus techniques liés à l'authentification dans le cadre de l'apprentissage de NextAuth pour les applications Next.js.

L'utilisation de ce contenu est strictement limitée à des fins d'apprentissage et de formation. Il est essentiel de respecter les droits d'auteur et les lignes directrices de partage de contenu établis par la plateforme YouTube et par le créateur du contenu, Melvynxdev. Pour toute utilisation au-delà du cadre éducatif privé, veuillez consulter la politique de droits d'auteur et obtenir l'autorisation nécessaire de la part du créateur de contenu original.

Je reconnais pleinement la propriété intellectuelle et l'effort investi par Melvynxdev dans la création de ce matériel éducatif et je m'engage à l'utiliser de manière responsable, en créditant l'auteur et en orientant les autres vers sa chaîne pour une expérience d'apprentissage complète.

Chaine youtube : https://www.youtube.com/@melvynxdev

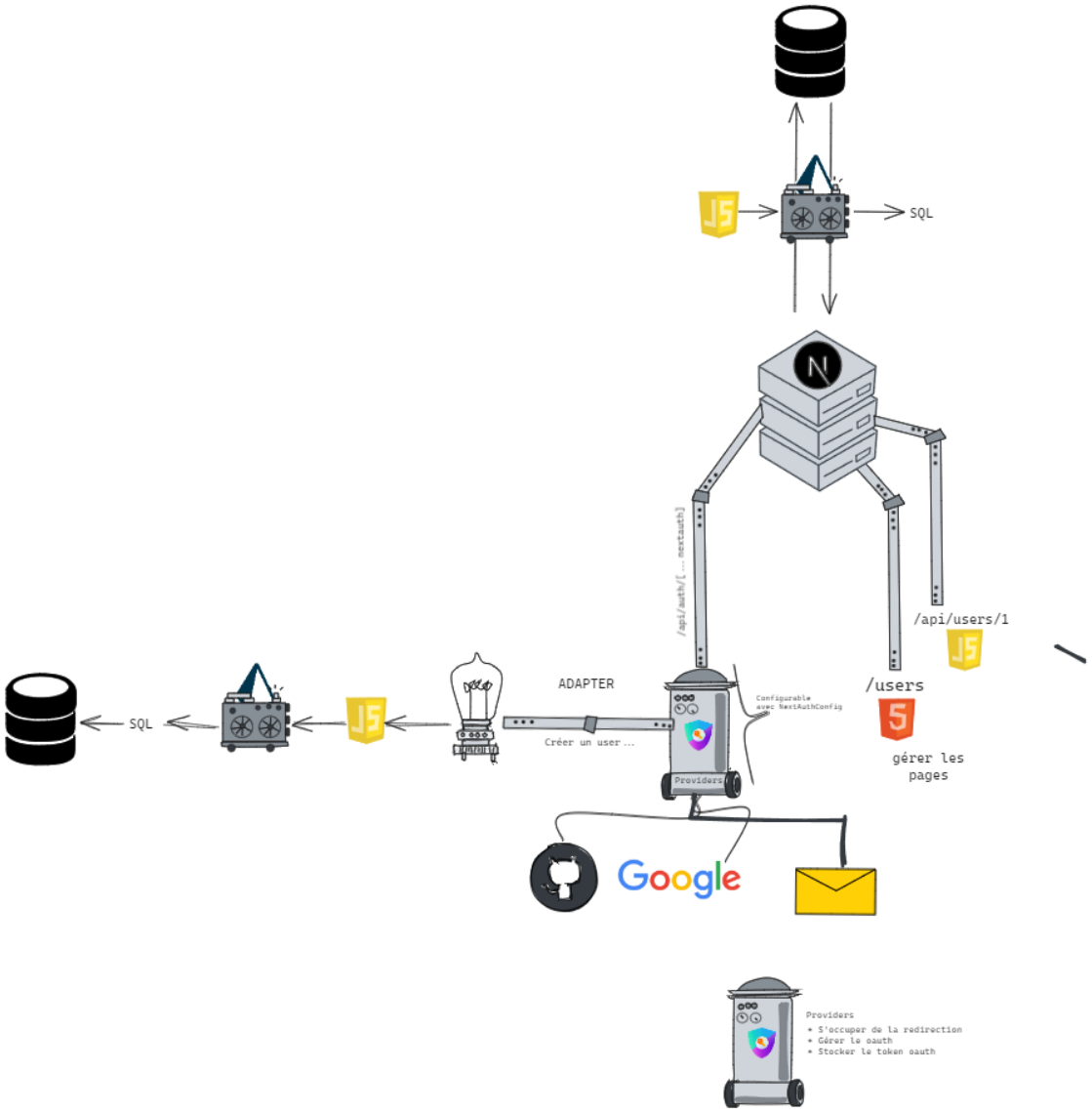
Lien vidéo nextAuth : https://www.youtube.com/watch?v=LaMoteg626I

Table des matières

Étape 1 : Comprendre le Fonctionnement de NextAuth

NextAuth est un complément pour Next.js. Il vise à simplifier les processus d'authentification et de gestion des sessions dans les applications développées avec Next.js. Voici les points clés de son fonctionnement :

1. **Intégration avec Next.js** : NextAuth s'intègre de manière transparente avec Next.js, permettant une mise en œuvre aisée de l'authentification dans les applications Next.js.
2. **Authentification et Gestion des Sessions** : NextAuth fournit des outils pour gérer l'authentification des utilisateurs et le maintien de leurs sessions.
3. **Prise en Charge de Divers Fournisseurs** : Il supporte plusieurs fournisseurs d'authentification tels que Google, Facebook, et bien d'autres, facilitant ainsi la connexion des utilisateurs via leurs comptes existants.
4. **Sécurité et Confidentialité** : NextAuth met l'accent sur la sécurité et la confidentialité, en assurant la protection des données des utilisateurs et la gestion sécurisée des sessions.
5. **Personnalisation et Extensibilité** : Il offre des options de personnalisation et est extensible pour répondre aux besoins spécifiques d'une application.
6. **Facilité d'Utilisation** : Conçu pour être simple et intuitif, NextAuth permet une mise en place rapide de l'authentification sans nécessiter une connaissance approfondie des systèmes d'authentification.



Fonctionnement de NextAuth dans Next.js

NextAuth fonctionne via le chemin **/api/auth/[…nextauth]** dans une application Next.js. Cette "machine" intégrée dans Next.js est conçue pour gérer l'authentification. Elle peut être configurée à l'aide de **NextAuthConfig** pour définir son mode de fonctionnement. Voici les aspects essentiels de cette configuration :

1. **Configuration avec NextAuthConfig** :
    - **NextAuthConfig** est utilisé pour personnaliser le comportement de NextAuth.
    - Il permet de définir les paramètres de fonctionnement, comme les méthodes d'authentification et les fournisseurs.
2. **Fournisseurs (Providers)** :
    - Le paramètre **Provider** est crucial dans NextAuth. Il sert à connecter différents services d'authentification.

- On peut le voir comme la connexion de "câbles" à divers services comme Google, GitHub, etc.

3. **Types de Fournisseurs (Providers)** :
    - **OAuth Provider** : Permet la connexion via des services comme Google, Facebook, etc.
    - **Credentials Provider** : Utilise une méthode standard basée sur le nom d'utilisateur et le mot de passe.
    - **Email Provider** :
        - Utilise le concept de "magic link" en se connectant à un serveur SMTP.
        - Permet l'authentification via un lien envoyé par email.
4. **Autres considérations** :
    - **HTTP-Based Email Provider** : Une variante de l'Email Provider qui utilise des protocoles basés sur HTTP.

Adapters dans NextAuth

Les adapters dans NextAuth sont des composants essentiels qui permettent d'intégrer NextAuth avec diverses bases de données ou systèmes de gestion de données. Voici les éléments clés à comprendre à propos des adapters dans NextAuth :

1. **Définition d'un Adapter** :
    - Un adapter est un objet qui contient des méthodes pour interagir avec une base de données ou un système de stockage de données.
    - Il joue le rôle d'intermédiaire entre NextAuth et la base de données, gérant la création, la mise à jour et la récupération des données d'utilisateurs.
2. **Paramètres d'un Adapter** :
    - **Client** : Le paramètre client fait référence à l'instance de connexion à la base de données ou au service de stockage de données.
    - **Options** : Les options permettent de personnaliser le comportement de l'adapter, comme la configuration des tables ou collections, ou des paramètres spécifiques au type de base de données utilisée.
3. **Méthodes d'un Adapter** :
    - Les adapters contiennent plusieurs méthodes pour effectuer diverses opérations liées à l'authentification et à la gestion des utilisateurs.
    - Ces méthodes incluent, mais ne sont pas limitées à, la création de sessions, la gestion des comptes utilisateurs, la mise à jour des informations de profil, et la validation des tokens.
4. **Personnalisation et Flexibilité** :

- Les adapters offrent une grande flexibilité et peuvent être personnalisés pour s'adapter à différents types de bases de données (comme MySQL, PostgreSQL, MongoDB, etc.).
- Ils sont essentiels pour assurer que NextAuth fonctionne correctement avec le système de stockage de données choisi.

```
import type { Adapter } from '@auth/core/adapters'

export function MyAdapter(client, options = {}): Adapter {
  return {
    async createUser(user) {
      return
    },
    async getUser(id) {
      return
    },
    async getUserByEmail(email) {
      return
    },
    async getUserByAccount({ providerAccountId, provider }) {
      return
    },
    async updateUser(user) {
      return
    },
    async deleteUser(userId) {
      return
    },
    async linkAccount(account) {
      return
    }
  }
}
```

Structure de l'Adaptateur Customisé

Chaque méthode définie dans l'adaptateur doit interagir avec la base de données pour effectuer des opérations spécifiques liées à l'authentification et à la gestion des utilisateurs :

- **createUser(user)**: Crée un nouvel utilisateur dans la base de données.
- **getUser(id)**: Récupère un utilisateur par son identifiant unique.
- **getUserByEmail(email)**: Obtient les détails d'un utilisateur par son adresse e-mail.
- **getUserByAccount({ providerAccountId, provider })**: Trouve un utilisateur en fonction de son identifiant de fournisseur et du nom du fournisseur.
- **updateUser(user)**: Met à jour les informations d'un utilisateur existant.
- **deleteUser(userId)**: Supprime un utilisateur de la base de données.

- **linkAccount(account)**: Associe un compte de fournisseur externe (comme Google ou Facebook) à un utilisateur.

Intégration avec Prisma

Prisma est utilisé comme un ORM qui facilite les interactions entre le code JavaScript et la base de données SQL :

1. **Transformation du Code JavaScript en SQL** :
    - Prisma convertit les appels de fonction JavaScript en requêtes SQL qui sont exécutées sur la base de données.
    - Cette abstraction permet aux développeurs de travailler avec la base de données en utilisant une syntaxe JavaScript familière sans écrire de SQL directement.
2. **Communication avec la Base de Données** :
    - Lorsqu'une méthode de l'adaptateur est appelée, Prisma génère la requête SQL correspondante.
    - La base de données exécute la requête et retourne les résultats.
3. **Retour de l'Utilisateur (user)** :
    - Par exemple, lorsque **createUser(user)** est appelée, Prisma envoie une instruction SQL pour insérer un nouvel utilisateur dans la base de données.
    - Une fois l'utilisateur créé, la base de données renvoie les détails de l'utilisateur à l'application.

L'adaptateur personnalisé doit être complété avec la logique spécifique pour interagir avec Prisma. Par exemple, dans **createUser(user)**, il faudrait écrire le code qui appelle Prisma pour créer un nouvel utilisateur dans la base de données, et ensuite retourner cet utilisateur ou l'identifiant de l'utilisateur créé. Il en va de même pour les autres méthodes, où chacune interagit avec la base de données via Prisma pour effectuer l'action demandée.

Étape 2 : Configuration de NextAuth avec Prisma

Voici les étapes détaillées pour configurer NextAuth avec Prisma dans un projet Next.js :

1. **Installer Prisma CLI :**
    - Commencer par installer Prisma CLI globalement ou localement dans le projet en utilisant npm ou yarn.

      npm install @prisma/cli --save-dev

      ou

      yarn add @prisma/cli --dev

2. **Installer le Client Prisma :**
    - Installer le client Prisma, qui est utilisé pour accéder à la base de données à travers Prisma.

      ```
      npm install @prisma/client
      ```

      ou

      ```
      yarn add @prisma/client
      ```

3. **Installer NextAuth :**
    - Procéder à l'installation de NextAuth dans le projet.

      ```
      npm install next-auth
      ```

      ou

      ```
      yarn add next-auth
      ```

4. **Créer le Dossier d'Authentification :**
    - Créer un dossier pour les routes d'authentification dans le projet Next.js sous **pages/api/auth**.
    - À l'intérieur de ce dossier, créer un fichier **[...nextauth].ts**. Ce fichier traitera les requêtes d'authentification.

5. **Créer le Fichier AuthConfig :**
    - À l'intérieur du fichier **[...nextauth].ts**, importer NextAuth et le configurer avec les options souhaitées.
    - Spécifier les fournisseurs, les callbacks, les sessions, les pages d'authentification personnalisées, etc.

6. **Adapter le Modèle Account :**
    - Copier le modèle Account fourni par NextAuth pour Prisma.
    - Adapter le modèle selon les besoins de la base de données en supprimant ou en modifiant les champs qui ne sont pas nécessaires ou qui doivent être personnalisés.

7. **Migrer la Base de Données :**

- Après avoir configuré les modèles dans le fichier schema.prisma, utiliser la commande Prisma pour migrer la base de données.

  ```
  npx prisma migrate dev --name init
  ```

  ou

  ```
  yarn prisma migrate dev --name init
  ```

Ces étapes aideront à mettre en place l'authentification dans une application Next.js en utilisant NextAuth et Prisma pour gérer les utilisateurs et leurs sessions de manière sécurisée et efficace. Il est important de suivre chaque étape avec soin et de consulter la documentation de NextAuth et Prisma pour des informations plus détaillées sur la configuration et les options disponibles.

1. Importer le Fournisseur GitHub : Ouvrir le fichier de configuration de NextAuth (habituellement **[...nextauth].ts** dans le dossier **pages/api/auth**) et importer le fournisseur GitHub depuis **next-auth/providers**.

2. Configurer le Fournisseur GitHub : Dans le même fichier de configuration, ajouter le fournisseur GitHub à la liste des fournisseurs dans la configuration de NextAuth. Il faudra l'ID client GitHub (**GITHUB_ID**) et le secret client GitHub (**GITHUB_SECRET**) qui sont obtenus en créant une nouvelle application OAuth sur GitHub.

3. Créer une Application OAuth sur GitHub :

    - Aller dans les paramètres du compte GitHub.
    - Rechercher la section "Developer settings" et choisir "OAuth Apps".
    - Cliquer sur "New OAuth App".
    - Remplir les détails nécessaires :
        - Nom de l'application : Le nom de l'application Next.js.
        - URL de la page d'accueil : L'URL du site (pour le développement local, cela peut être **http://localhost:3000**).
        - URL de rappel d'autorisation : L'URL où les utilisateurs sont redirigés après l'authentification (pour le développement local, cela peut être **http://localhost:3000/api/auth/callback/github**).

4. Autoriser localhost:3000 :

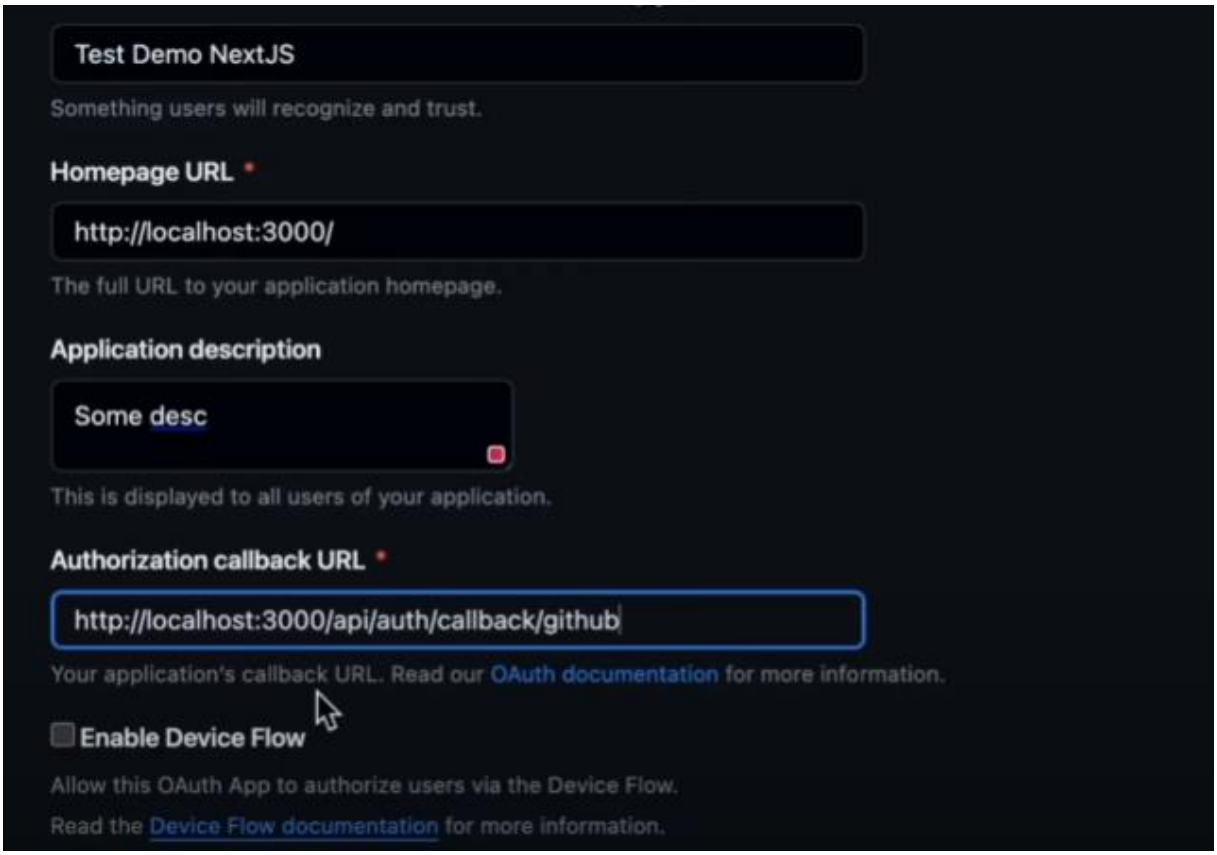
    - Lors de la création de l'application OAuth sur GitHub, ajouter **http://localhost:3000** et **http://localhost:3000/api/auth/callback/github** aux champs "Homepage URL" et "Authorization callback URL" respectivement. Cela permettra à GitHub d'authentifier les requêtes provenant de l'environnement de développement local.

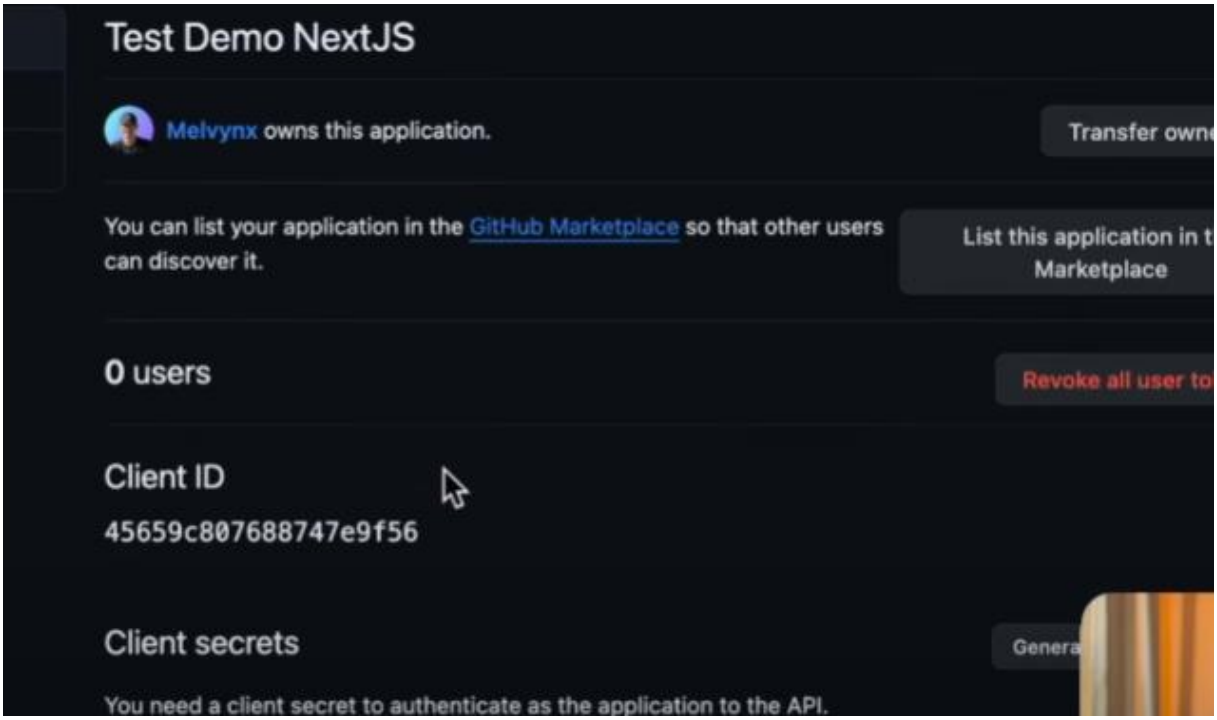
5. Ajouter les Variables d'Environnement :

    - Ajouter les variables d'environnement **GITHUB_ID** et **GITHUB_SECRET** au fichier **.env.local** pour l'environnement de développement local.
    - Ces variables doivent correspondre aux valeurs de l'ID client et du secret client reçues de GitHub lors de la création de l'application OAuth.

6. Redémarrer le Serveur de Développement :

    - Après avoir configuré le fournisseur GitHub et mis à jour les variables d'environnement, redémarrer le serveur de développement pour que les changements prennent effet.



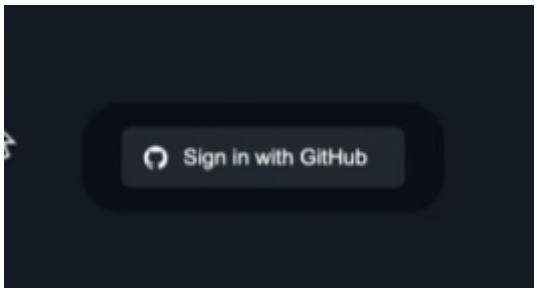


Ce composant LoginButton peut être utilisé n'importe où dans l'application pour permettre aux utilisateurs de se connecter. La fonction signIn sans argument redirige l'utilisateur vers la page de connexion par défaut de NextAuth où il peut choisir parmi les fournisseurs d'authentification

configurés. Pour rediriger directement vers un fournisseur spécifique comme GitHub, il est possible de passer un argument à **signIn('github').**

```
import { signIn } from 'next-auth/react';

export const LoginButton = () => {
  return (
    <button
      onClick={async () => {
        await signIn();
      }}
      className="btn btn-primary"
    >
      Login
    </button>
  );
};
```

Ensuite lorsque nous appuyons sur login cette page s'affiche :



Comment savoir si on est authentifié :

Dans home

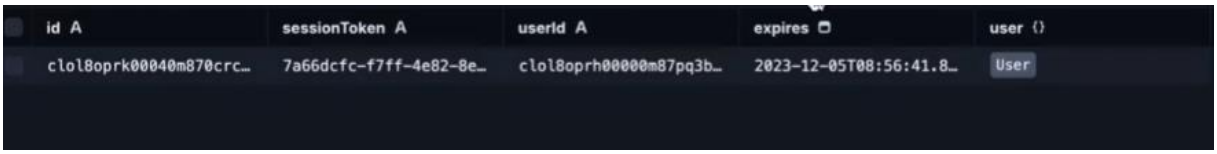
Const session : →

```
export default async function Home() {
  const session = await getServerSession(authConfig);

  if (session) {
    return <pre>{JSON.stringify(session, null, 2)}</pre>;
  }

  return (
    <div>
      <LoginButton />
    </div>
  );
}
```

- **getServerSession(authConfig)**: Cette fonction est appelée pour obtenir la session actuelle de l'utilisateur. Elle doit être configurée avec les options d'authentification définies pour l'application.
- **if (session) { ... }**: On vérifie si l'objet session existe. S'il existe, cela signifie que l'utilisateur est actuellement authentifié et il est possible d'utiliser les données de session pour afficher des informations utilisateur ou personnaliser l'interface.
- **return <pre>{JSON.stringify(session, null, 2)}</pre>**: Si une session est présente, les détails de la session sont affichés au format JSON pour pouvoir les voir sur la page. Le **pre** est utilisé pour conserver la mise en forme du JSON.
- **return <LoginButton /> **: Si aucune session n'est présente (c'est-à-dire que session est **undefined** ou **null**), un bouton de connexion est affiché qui permet à l'utilisateur de se connecter.

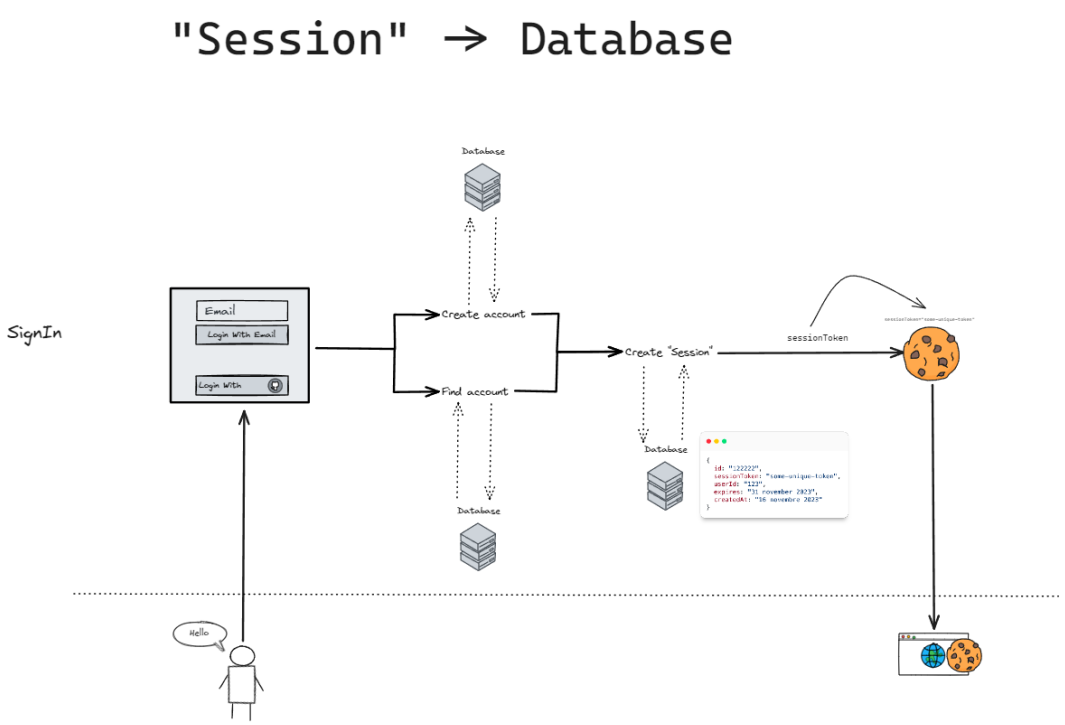
Pour que ce code fonctionne, il faut avoir configuré NextAuth correctement et fourni la configuration nécessaire (**authConfig**) à la fonction **getServerSession**. Il est également nécessaire de s'assurer que le composant **LoginButton** est bien importé dans le fichier où ce code est utilisé.

C’est quoi « session » :



Dans le contexte de NextAuth, une session représente un état d'authentification actif pour un utilisateur. La capture d'écran partagée montre les détails typiques stockés pour une session d'utilisateur. Voici ce que chaque champ représente généralement :

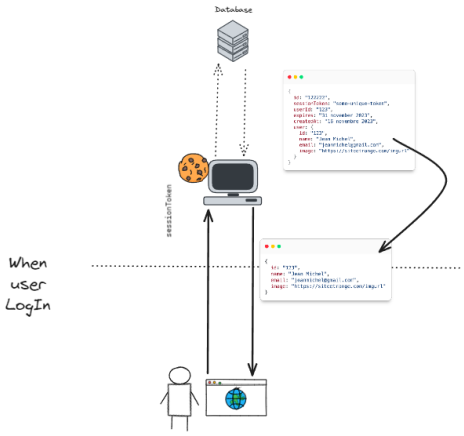
1. **id**: C'est un identifiant unique pour chaque session. Il peut servir à référencer une session spécifique dans la base de données ou le stockage de sessions.
2. **sessionToken**: C'est un token ou une chaîne de caractères cryptographique utilisée pour authentifier une session d'utilisateur lors des interactions avec le serveur. Le token de session est envoyé avec chaque requête pour vérifier l'authentification de la session.
3. **userId**: C'est l'identifiant unique de l'utilisateur associé à cette session. Cela permet de relier la session à un utilisateur spécifique dans la base de données des utilisateurs.
4. **expires**: Cette valeur indique la date et l'heure auxquelles la session expirera. Passé ce délai, la session ne sera plus valide et une nouvelle connexion sera nécessaire.

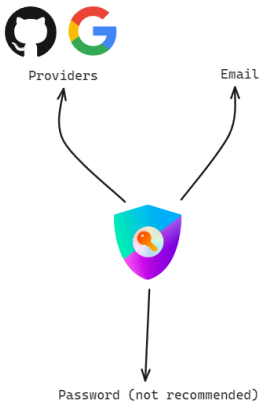


voici ce qui se passe généralement lorsqu'une session est créée :

1. **Association Session-Utilisateur** :
    - Lorsqu'un utilisateur se connecte avec succès, NextAuth crée une session qui associe un **sessionToken** unique à l'utilisateur.
    - Un enregistrement de session est créé dans votre stockage de données, liant le **sessionToken** à un **userId**.
2. **Stockage de SessionToken** :
    - Le **sessionToken** est stocké dans un cookie sur le navigateur de l'utilisateur.
    - Ce cookie est envoyé avec chaque requête HTTP au serveur, ce qui permet à NextAuth d'authentifier la requête en validant le **sessionToken**.
3. **Sécurité** :
    - Les cookies utilisés par NextAuth sont généralement configurés comme des cookies HTTPOnly pour des raisons de sécurité, ce qui signifie qu'ils ne sont pas accessibles via JavaScript côté client.
    - Cela aide à prévenir certaines attaques telles que le cross-site scripting (XSS).
4. **Expiration** :
    - Le cookie a également une date d'expiration, qui est généralement synchronisée avec le champ **expires** de l'enregistrement de session dans la base de données.
    - Lorsque le cookie expire, l'utilisateur doit se reconnecter pour obtenir un nouveau **sessionToken**.

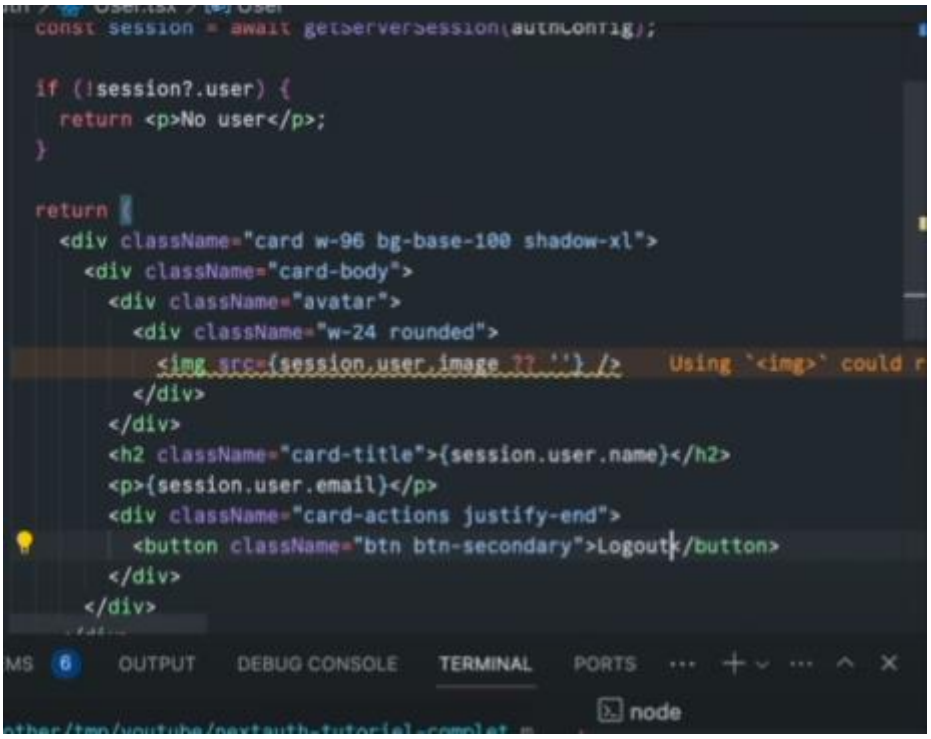
La gestion des sessions est cruciale pour la sécurité de l'authentification utilisateur dans les applications web modernes. Elle permet de s'assurer que les utilisateurs sont bien ceux qu'ils prétendent être et empêche l'accès non autorisé aux comptes utilisateurs.





Voici comment cela fonctionne et les implications de la suppression du cookie :

1. **Utilisation du Cookie avec SessionToken** :
    - Chaque fois que l'utilisateur effectue une requête vers le serveur, le navigateur envoie automatiquement le cookie contenant le **sessionToken**.
    - Le serveur, via NextAuth, vérifie le **sessionToken** contre les enregistrements de session stockés pour valider l'authentification de l'utilisateur.
2. **Rôle du Cookie dans l'Authentification** :
    - Le cookie sert de preuve d'authentification. Tant que le token est valide et que le cookie n'est pas expiré, l'utilisateur est considéré comme authentifié.
3. **Suppression du Cookie** :
    - Si l'utilisateur supprime manuellement le cookie ou si le cookie est supprimé par le navigateur (par exemple, à l'expiration ou en nettoyant les données de navigation), la session côté client est terminée.
    - La prochaine fois que l'utilisateur tentera d'accéder à une partie sécurisée de l'application, il sera invité à se reconnecter, car le serveur ne pourra pas valider la session sans le **sessionToken**.
4. **Reconnexion Nécessaire** :
    - Lorsque l'utilisateur se reconnecte, NextAuth génère un nouveau **sessionToken** et met à jour le cookie dans le navigateur de l'utilisateur.
    - Ceci établit une nouvelle session, permettant à l'utilisateur de continuer à utiliser l'application sans interruption.



Pour permettre aux utilisateurs de se déconnecter de l'application Next.js utilisant NextAuth, il faut effectuer les étapes suivantes :

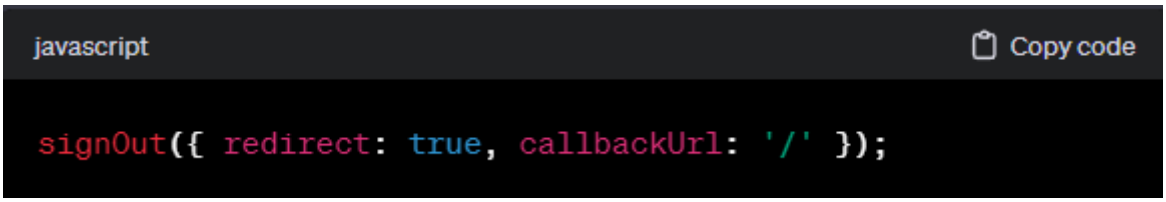
1. Créer un Composant ou une Fonction de Déconnexion : Il est possible de créer un nouveau composant ou une fonction qui appellera la méthode **signOut** fournie par NextAuth.
2. Appeler la Méthode **signOut** : La méthode **signOut** va effacer la session de l'utilisateur et supprimer le cookie de session, ce qui déconnectera l'utilisateur.

Voici comment il est possible de créer un bouton de déconnexion qui utilise la méthode **signOut** :

javascript

```javascript
// Importez la méthode signOut de next-auth/react
import { signOut } from 'next-auth/react';

// Définissez un composant bouton qui appelle signO
export const LogoutButton = () => {
  return (
    <button onClick={() => signOut()}>
      Logout
    </button>
  );
};
```

Lorsque le bouton est cliqué, l'utilisateur sera déconnecté. Il est également possible de personnaliser le comportement de la méthode **signOut** en passant des options. Par exemple, il est possible de rediriger l'utilisateur vers la page d'accueil après la déconnexion :



Dans ce cas, après la déconnexion, l'utilisateur sera redirigé vers la racine de l'application ('/'). Note : Il est important de s'assurer que le composant **LogoutButton** est utilisé dans les parties de l'application où il est souhaité offrir une option de déconnexion, comme dans la barre de navigation ou dans un menu utilisateur.

```
import { signOut } from 'next-auth/react';

export const LogoutButton = () => {
  return (
    <button
      onClick={async () => {
        await signOut();
      }}
      className="btn btn-primary"
    >
      Logout
    </button>
  );
};
```

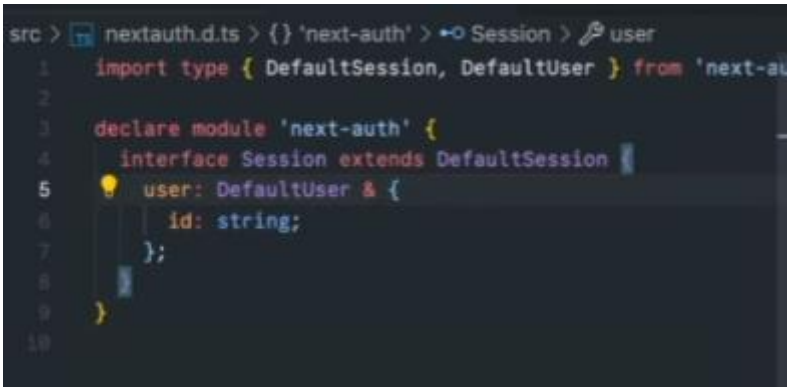
Dans NextAuth, les callbacks permettent de personnaliser le comportement pendant le processus d'authentification. Pour ajouter des informations supplémentaires à la session, comme l'ID de l'utilisateur, il est possible de le faire en utilisant un callback de session dans la configuration NextAuth.

Dans cet exemple, le callback session reçoit deux arguments : **session** et **user**. L'objet **session** est ce qui est renvoyé par NextAuth à l'application front-end, tandis que **user** est l'objet utilisateur tel qu'il est stocké dans la base de données.

Le callback de session modifie l'objet de session pour inclure l'ID de l'utilisateur, qui est ensuite accessible dans l'application client.

Important : Il faut s'assurer de ne pas partager d'informations sensibles ou privées par le biais de la session qui pourraient être exploitées si elles étaient interceptées par une partie non autorisée. L'ID de l'utilisateur est généralement considéré comme non sensible et est souvent nécessaire côté client pour les requêtes de données ou la gestion de l'état utilisateur.

L'id n'existe pas, pour ça il faut créer un fichier nextauth.d.js



La capture d'écran montre un fichier de déclaration TypeScript qui étend les types par défaut de NextAuth pour inclure un "id" dans l'objet "user" d'une session. Cela permet de typer correctement la session dans tout le projet, en s'assurant que l'ID de l'utilisateur est reconnu comme faisant partie de l'objet de session.

Pour utiliser l'ID de l'utilisateur dans votre projet Next.js avec NextAuth, il faut suivre ces étapes :

1. Créer un fichier de déclaration de types :
    - Nommer ce fichier "next-auth.d.ts" (et non ".js") pour qu'il soit reconnu comme un fichier de déclaration TypeScript.
2. Étendre les Types de Session et d'Utilisateur :
    - Dans ce fichier, étendre les types par défaut pour inclure l'ID de l'utilisateur.

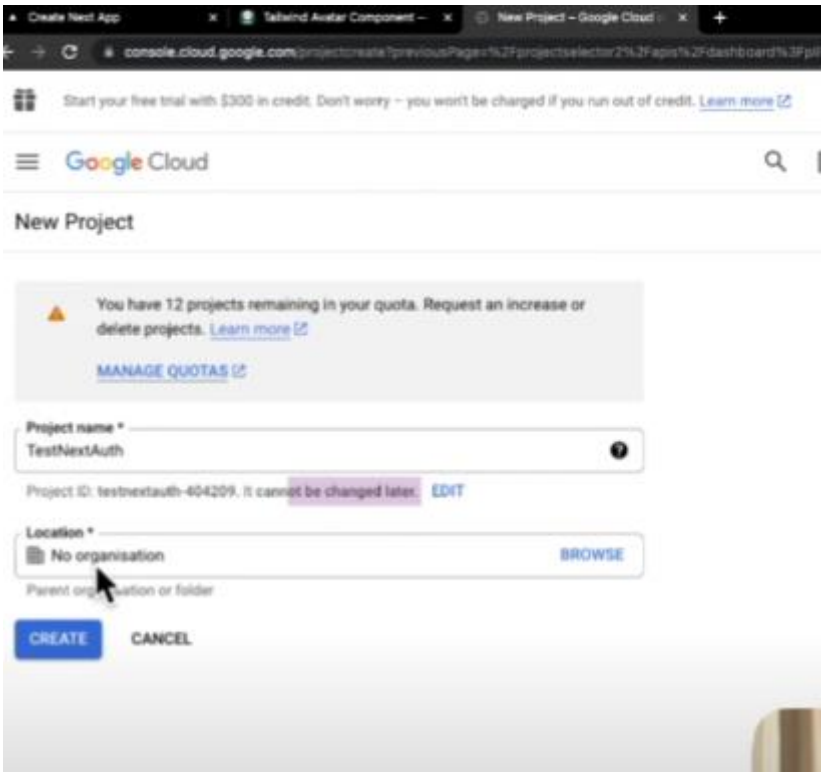
Après avoir ajouté cette déclaration, il est possible d'utiliser "session.user.id" dans votre application avec la certitude que TypeScript reconnaîtra "id" comme faisant partie de l'objet "user".

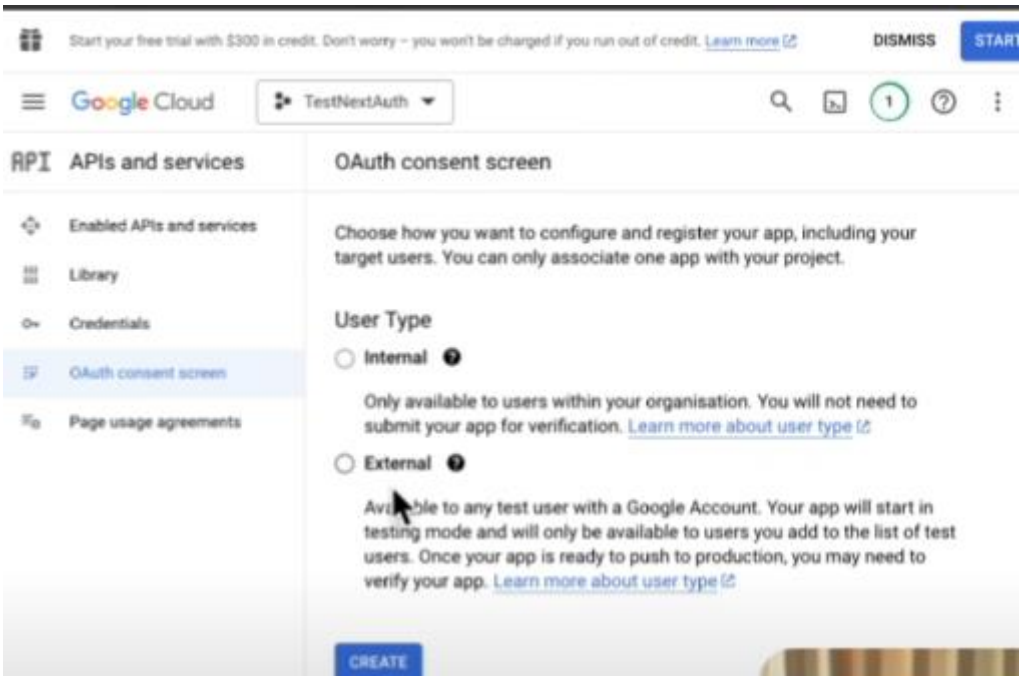
Note : Il est important de s'assurer que le fichier de déclaration de types est correctement importé ou accessible dans votre projet TypeScript. Habituellement, TypeScript reconnaîtra automatiquement les fichiers ".d.ts" s'ils se trouvent dans votre répertoire de projet ou si les chemins sont correctement configurés dans votre fichier "tsconfig.json".

SETUP GOOGLE

Pour configurer Google OAuth dans votre application Next.js avec NextAuth, il faut suivre les étapes suivantes pour mettre en place l'authentification avec Google :

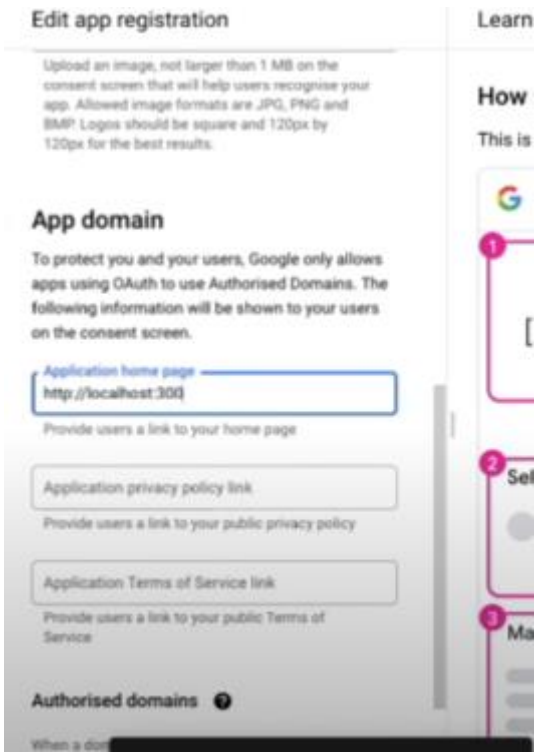
1. Créer un Projet et Configurer l'OAuth 2.0 dans la Console Google Cloud :
    - Accéder à la console Google Cloud.
    - Créer un nouveau projet ou sélectionner un projet existant.
    - Aller dans "APIs & Services" > "Credentials".
    - Cliquer sur "Create Credentials" et sélectionner "OAuth client ID".
    - Configurer l'écran de consentement OAuth si nécessaire.
    - Ajouter http://localhost:3000/api/auth/callback/google à la liste des URI de redirection autorisés si vous êtes en phase de développement.
    - Une fois les étapes terminées, vous obtiendrez un **Client ID** et un **Client Secret**.





Cliquer sur external

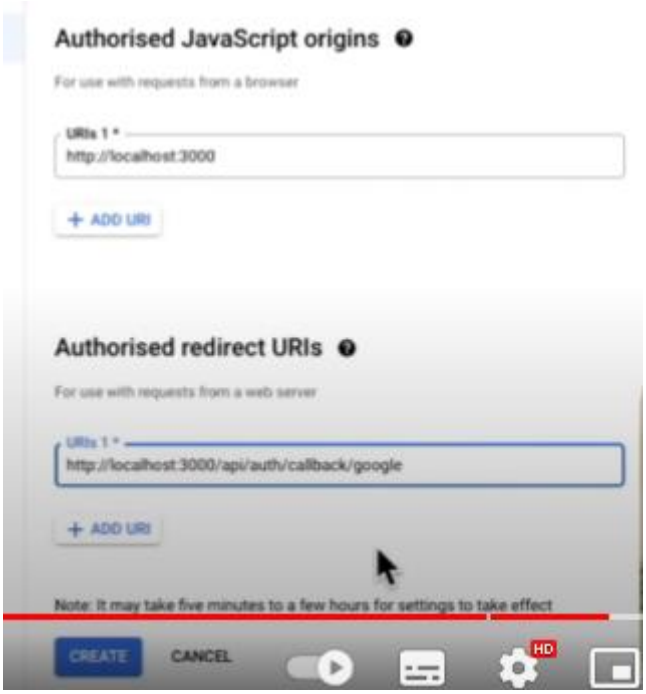
Remplir les informations pour comme pour github :



Pour créer des Identifiants OAuth 2.0 :

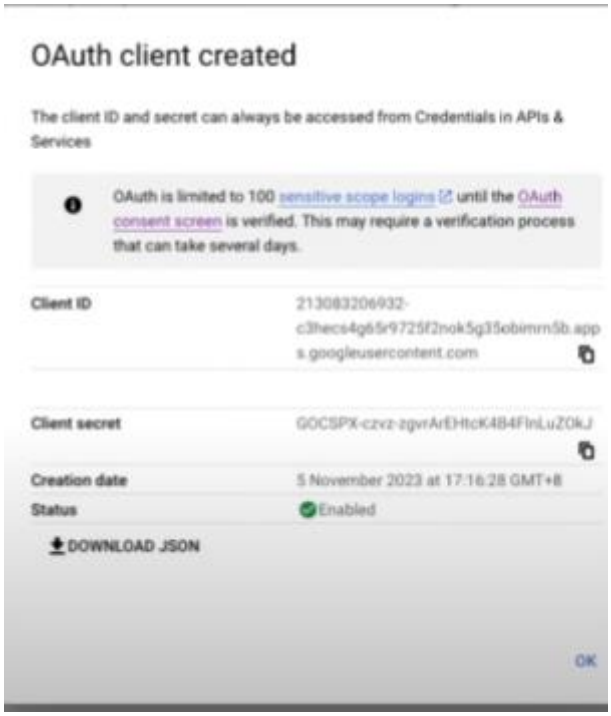
- Accéder à la section "Credentials" dans "APIs & Services".

- Cliquer sur "Create credentials" et sélectionner "OAuth client ID".
- Suivre les instructions pour configurer l'ID client OAuth. Assurer de inclure les URI de redirection autorisés pour votre application, tels que http://localhost:3000/api/auth/callback/google pour le développement local.



Obtenir le Client ID et le Client Secret :

Après avoir créé les identifiants, un Client ID et un Client Secret seront fournis pour l'application.



Mettre à Jour le Fichier .env :

- Ajouter le **Client ID** et le **Client Secret** dans votre fichier **.env.local** à la racine de votre projet Next.js.
- Importer Google Provider

Configuration par mail

Pour configurer l'authentification par email en utilisant un serveur SMTP dans NextAuth, il faut effectivement configurer les informations du serveur SMTP dans le fichier **.env** du projet Next.js. Voici comment il est possible de le faire, en prenant l'exemple d'un service SMTP appelé "Resend" :

1. Configurer un Serveur SMTP :
    - Se rendre sur le site du fournisseur SMTP, par exemple Resend, et créer un compte ou utiliser un existant.
    - Configurer un serveur SMTP et obtenir les informations de connexion nécessaires : le nom d'utilisateur, le mot de passe, l'hôte et le port.
2. Ajouter les Informations SMTP dans le Fichier **.env** :
    - Dans le projet Next.js, ajouter les informations de connexion SMTP au fichier **.env.local** :

```
SMTP_USER="resend"
SMTP_PASSWORD="re_Dyr29AqG_P82BW4PhCjmgqeF2d9wc5eYv"
SMTP_HOST="smtp.resend.com"
SMTP_PORT="587"
EMAIL_FROM="contact@codeline.app"
```

Avec cette configuration, NextAuth utilisera le serveur SMTP pour envoyer des emails, par exemple pour l'authentification par "magic link". Il est important de bien protéger les informations de connexion SMTP et de ne pas les inclure dans le contrôle de version du code source.