

API REST

express/node js

Documentation de l'api rest

Avoir installé npm sur son poste avec ce lien <https://nodejs.org/fr/download>
installer pnpm avec 'npm i -g pnpm'

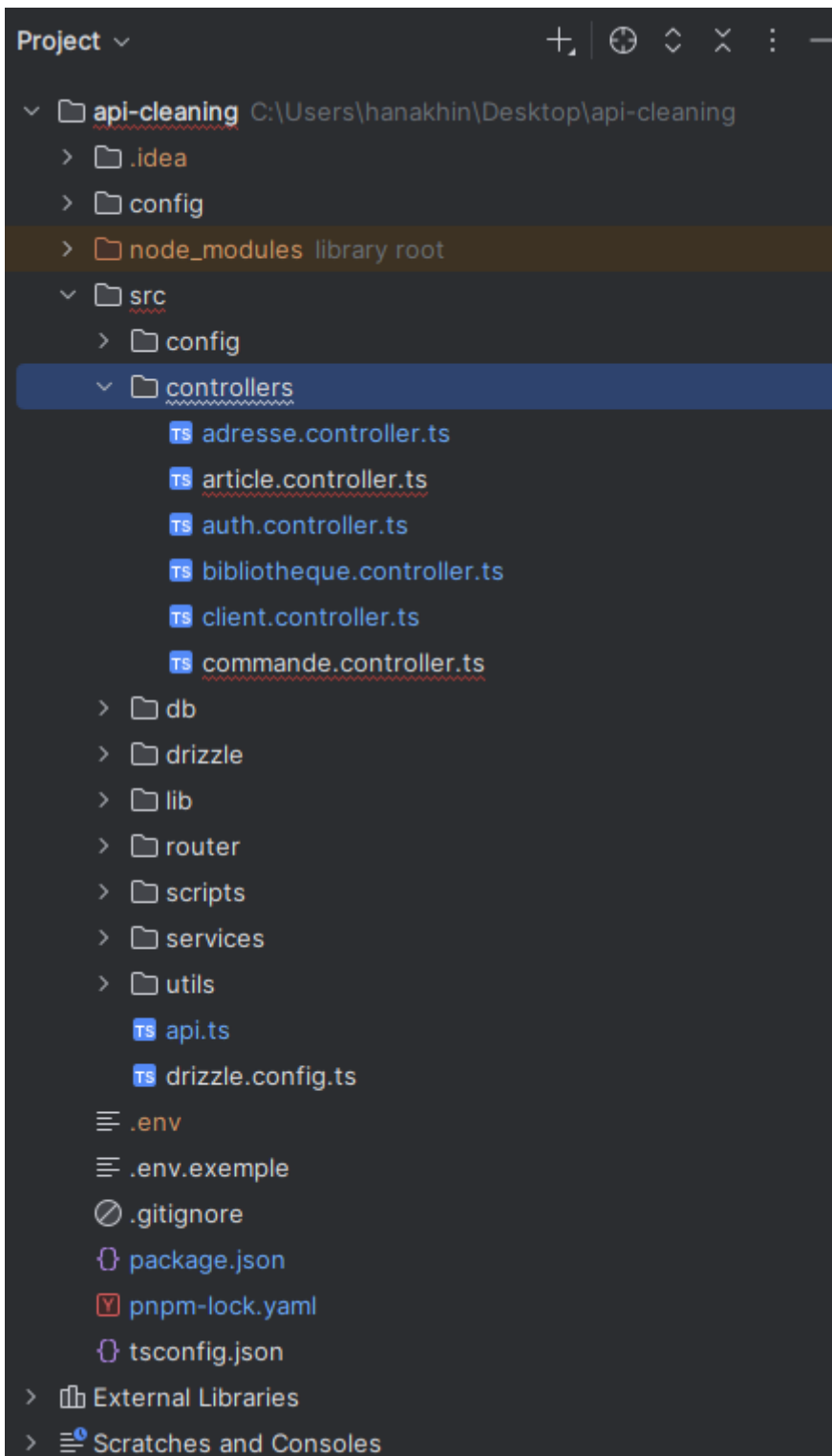
lancer le serveur de l'api avec la commande presente dans le package.json a la racine du projet

exemple : 'pnpm start' pour l'api E-CARE cleaning

- [E-CARE API CLEANING](#)
- [Créer une route](#)
- [Tester sa route avec postman](#)

E-CARE API CLEANING

Processus de création d'une route -> controller -
> service



1)Créer le controller

```

1  import {clientService} from "../services/client.service";
2  import {getDb} from "../lib/mssql";
3  import {sendError, sendSuccess} from "../utils/helpers/response.helper";
4
5  export const clientController = { Show usages  & hanakhin *
6
7      all: async (req:any,res:any) : Promise<Response<any, Record<string, any>... => {
8          try {
9              const pool : ConnectionPool = getDb(Number( value: 1))
10             const result : { cli: IResult<any>; cde: IResult<any>} = await clientService.getAll(pool)
11             return sendSuccess(res, result, status: 200)
12         } catch (err) {
13             console.log(err)
14             return sendError(res, err, status: 500)
15         }
16     },
17 }

```

un controller contient tout ce qui est en rapport avec les requetes/responses, c'est ici qu'on fait le try catch et qu'on renseigne par exemple le pool (db) etc .

on oublie pas de le rendre exportable avec le "export" const , et ensuite on crée nos methodes.

On peut le faire de plusieurs façons comme vu ici

```

import {clientService} from "../services/client.service";
import {getDb} from "../lib/mssql";
import {sendError, sendSuccess} from "../utils/helpers/response.helper";

export const clientController = { Show usages  & hanakhin *
>
    all: async (req:any,res:any) : Promise<Response<any, Record<string, any>... => {...},
}
export async function all (req:any,res:any) : Promise<void> {} no usages new *
export const all2 : () => void = () : void => {} no usages new *

```

trois méthodes différentes qui font la même chose , question de préférence.

donc dans le controller on appelle le service clientService.[la méthode qu'on veut] ([le paramètre que la méthode attend]).

```
const result : { cli: IResult<any>; cde: IResult<any>} = await clientService.getAll(pool)
```

2)Créer le service

```
export const clientService = { Show usages & hanakhin *
  getAll: async (pool: mssql.ConnectionPool) : Promise<cli: IResult<any>; cde: IResult... => {
    const cliResult :IResult<any> = await pool.request().query( command: `
      SELECT TOP 10 cli.*
      FROM dbo.cli AS cli
      LEFT JOIN dbo.tst AS tst ON tst.ctiers = COALESCE(NULLIF(cli.code, ''), cli.censeigne)
      LEFT JOIN dbo.tsd AS tsd ON tsd.ctiers = COALESCE(NULLIF(cli.code, ''), cli.censeigne)
    `);
    const cdeResult :IResult<any> = await pool.request().query( command: `
      SELECT TOP 10 cde.*, vtl.*
      FROM dbo.cde AS cde
      INNER JOIN dbo.vtl AS vtl ON vtl.numero = cde.numero
      INNER JOIN dbo.cli AS cli ON cli.code = cde.cclient
      WHERE cde.categorie = 'F'
    `);
    return {'cli':cliResult,'cde': cdeResult};
  }
}
```

dans le service on met tout ce qui est appel en db, et on retourne le resultat, le controller ce charge de la gestion d'erreur etc .

On fait toujours comme ca :

```
const cdeResult :IResult<any> = await pool.request().query( command: `
  SELECT TOP 10 cde.*, vtl.*
  FROM dbo.cde AS cde
  INNER JOIN dbo.vtl AS vtl ON vtl.numero = cde.numero
  INNER JOIN dbo.cli AS cli ON cli.code = cde.cclient
  WHERE cde.categorie = 'F'
`);
```

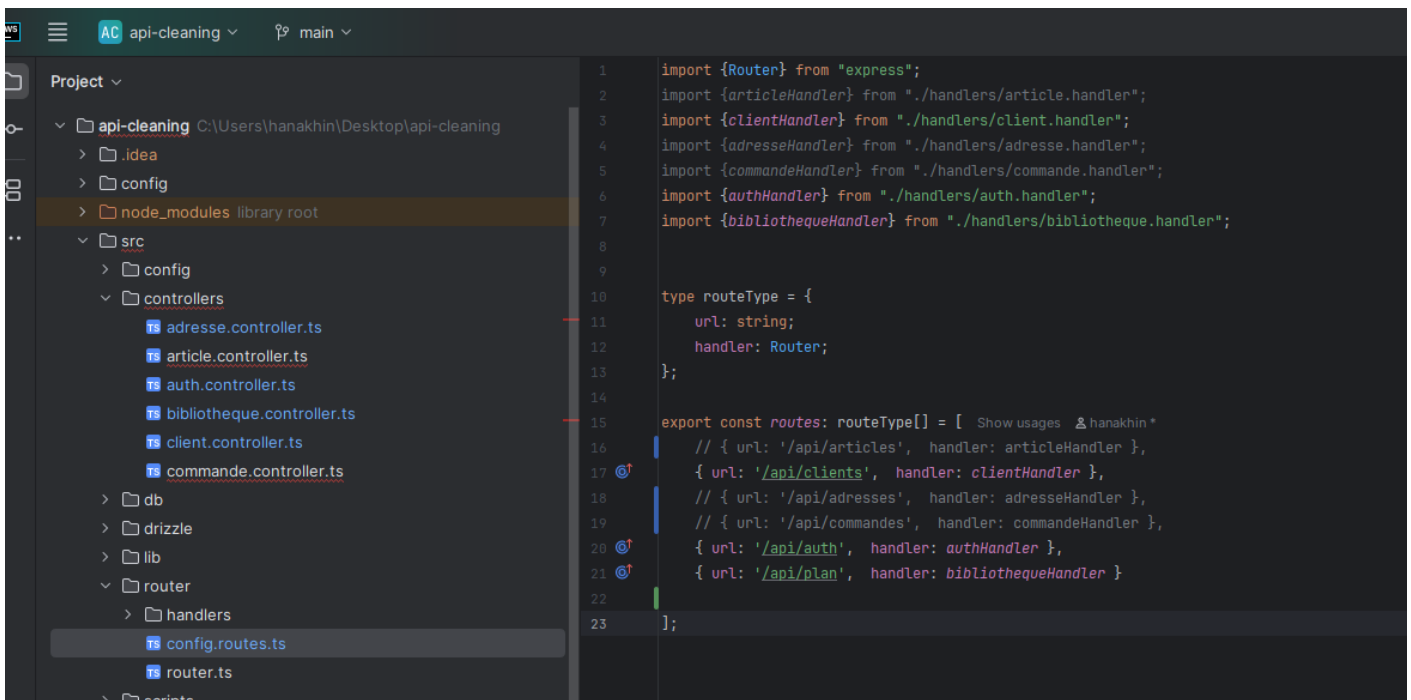
const [nom] = await (pour l'asynchrone) pool(parametre qu'on a typé).request().query(`[notre requete]`);

et on return le resultat

```
return {'cli':cliResult,'cde': cdeResult};
```

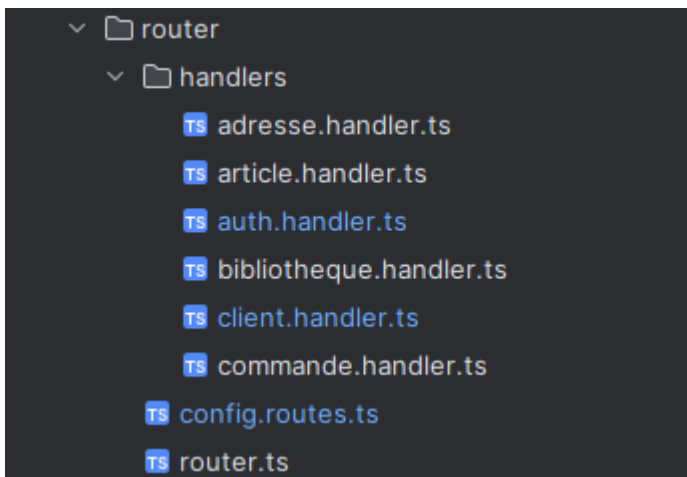
3)Créer la route

dans src/router/config.routes.ts on peut voir un tableau de routes, on duplique une des lignes et on change les valeurs par la route qu'on veut , url = [route de base], handler = [le router qui va gerer les routes]



```
1 import {Router} from "express";
2 import {articleHandler} from "../handlers/article.handler";
3 import {clientHandler} from "../handlers/client.handler";
4 import {adresseHandler} from "../handlers/adresse.handler";
5 import {commandeHandler} from "../handlers/commande.handler";
6 import {authHandler} from "../handlers/auth.handler";
7 import {bibliothequeHandler} from "../handlers/bibliotheque.handler";
8
9
10 type routeType = {
11   url: string;
12   handler: Router;
13 };
14
15 export const routes: routeType[] = [ Show usages & hanakhin "
16 // { url: '/api/articles', handler: articleHandler },
17 { url: '/api/clients', handler: clientHandler },
18 // { url: '/api/adresses', handler: adresseHandler },
19 // { url: '/api/commandes', handler: commandeHandler },
20 { url: '/api/auth', handler: authHandler },
21 { url: '/api/plan', handler: bibliothequeHandler }
22 ];
23 ];
```

ensuite dans router/handlers on crée un handler, [le nom qu'on veut].handler.ts



- router
 - handlers
 - adresse.handler.ts
 - article.handler.ts
 - auth.handler.ts
 - bibliotheque.handler.ts
 - client.handler.ts
 - commande.handler.ts
 - config.routes.ts
 - router.ts

```
1 import {Router} from "express";
2 import {clientController} from "../../controllers/client.controller";
3
4 export const clientHandler : Router = Router(); Show usages & hanakhin
5
6 clientHandler
7   .get( path: "/", clientController.all)
8
```

Dans ce handler, on crée notre export const [nom du handler] = Router() (crée une instance du router d'express.JS)

puis on appelle ce router fraîchement crée pour lui définir ses endpoints , ici on voit .get , ça crée donc la route /api/clients/ , en method GET , qui executera le code du controller qu'on mettra en second parametre (ici clientController.all)

avec tout ça , on a crée notre route d'api /api/clients qui execute la methode all du controller .

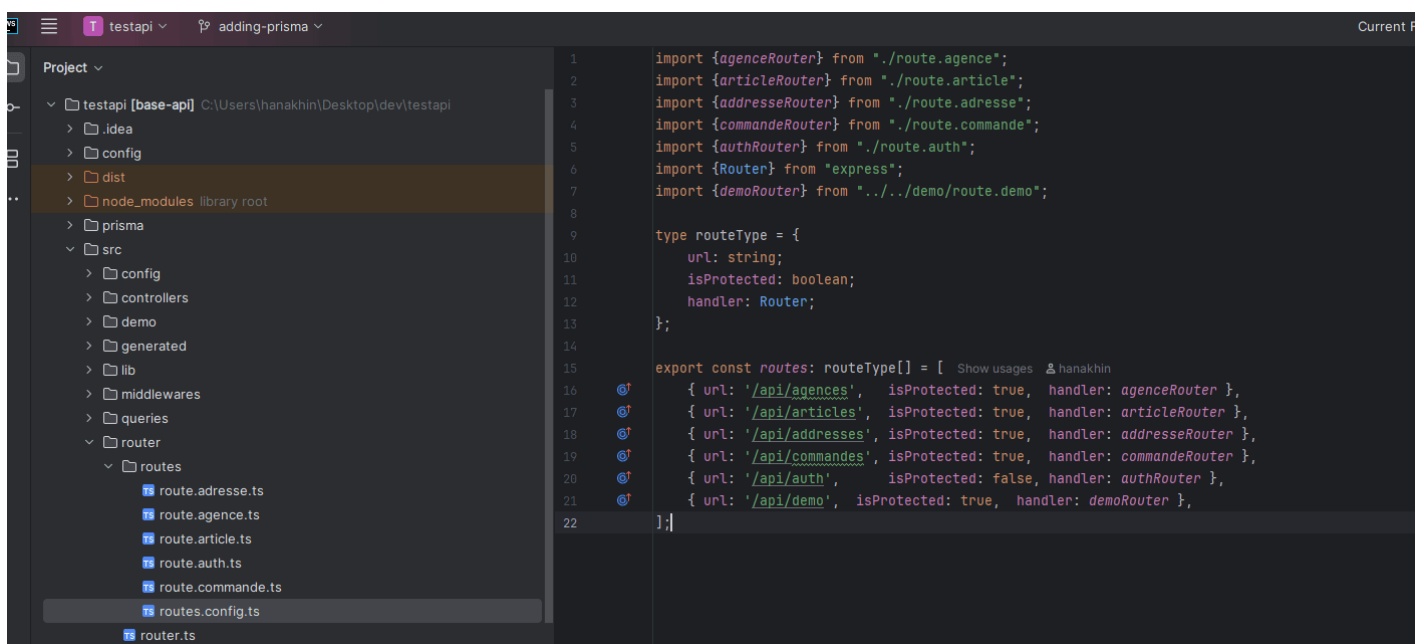
Pour tester on peut soit faire la route 'http://localhost:8000/api/clients' en get sur postman , ou simplement aller sur une page internet et mettre cette url dans la barre de recherche.

Créer une route

Pour créer une route il faut aller dans `src/router/routes/routes.config.ts` et y ajouter sa route dans le tableau de routes. Une route attend 3 paramètres :

L'URL en string, Le Booleen "isProtected" qui définit si la route est accessible hors connexion ou non , et le router qui lui est associé.

Le paramètre "url" définit la route de base du router associé , donc par exemple si on met `"/api/test"` dans url et le router "testRouter" dans le handler; toutes les routes du router commenceront par `"/api/test"`



```
1 import {agenceRouter} from "../route.agence";
2 import {articleRouter} from "../route.article";
3 import {adresseRouter} from "../route.adresse";
4 import {commandeRouter} from "../route.commande";
5 import {authRouter} from "../route.auth";
6 import {Router} from "express";
7 import {demoRouter} from "../../demo/route.demo";
8
9 type routeType = {
10   url: string;
11   isProtected: boolean;
12   handler: Router;
13 };
14
15 export const routes: routeType[] = [ Show usages & hanakhin
16   { url: '/api/agences', isProtected: true, handler: agenceRouter },
17   { url: '/api/articles', isProtected: true, handler: articleRouter },
18   { url: '/api/adresses', isProtected: true, handler: adresseRouter },
19   { url: '/api/commandes', isProtected: true, handler: commandeRouter },
20   { url: '/api/auth', isProtected: false, handler: authRouter },
21   { url: '/api/demo', isProtected: true, handler: demoRouter },
22 ];
```

une fois que c'est fait , il faut aller créer le router associé , on va utiliser le "DemoRouter" pour cet exemple.

dans le fichier `"src/router/routes"` on crée un fichier ts qu'on appelle `route.[le nom du router].ts`

```

1 import {Router} from "express";
2 import {CommandeController} from "../../controllers/commande.controller";
3
4 export const commandeRouter :Router = Router(); Show usages  👤 hanakhin
5
6 commandeRouter
7   .get( path: "/", CommandeController.getAllCommandesByCcli)
8   .post( path: "/", CommandeController.createPcde)
9

```

ensuite on crée une instance de Router qui est importé depuis "express"

puis , on définit les routes du router, donc comme on a dit avant toutes les routes du demoRouter on par défaut le prefix "/api/demo", donc il nous suffit de lui mettre l'endpoint qu'on veut précédé de la méthode de la route.

dans notre cas ici on a une get et une post , suivi du controlleur, et de la méthode qu'on veut utiliser.

L'étape suivante est de créer le controlleur associé comme ci dessous.

```

1 import { RequestHandler } from "express";
2 import { demoService } from "../demo.service";
3
4 export const DemoController :Record<string, RequestHandler>= { Show usages  👤 hanakhin *
5   getAllData: async (req: any, res: any) :Promise<any> => {
6     try {
7       const prisma :any = req.prisma;
8       const data :{code: number; reference: string | null;...} = await demoService.findAll(prisma);
9       return res.status(200).json(data);
10    } catch (err: any) {
11      console.error( message: "Error fetching items:", err);
12      return res.status(500).json({ error: err.message });
13    }
14  },
15
16  addData: async (req: any, res: any) :Promise<any> => {...},
17  deleteData: async (req: any, res: any) :Promise<any> => {...}
18 };
19

```

le controlleur contient la méthode qui permet d'aller chercher le service qui communique avec la base de donnée , lui il se charge de faire l'appel du service et de retourner une réponse .

soit une 200 si la requête est acceptée , soit une 500 si il y a un souci, soit une 401 si pas autorisé ... etc , ici on gère uniquement les services.

on crée ce contrôleur dans "src/controller" et on l'importe dans le router.

La dernière étape est de créer le service.

```
1  import { PrismaClient } from '../generated/client/client';
2  import { DemoDataType } from './demo.type';
3
4  export const demoService = { Show usages  hanakhin *
5
6  findAll: async (prisma: PrismaClient) : Promise<{ code: number; reference: string... => {
7    return prisma.cf1.findMany();
8  },
9
10 add: async (prisma: PrismaClient, data: DemoDataType) : Promise<{ code: number; reference: string... => {
11   return prisma.cf1.create({
12     data: {
13       code: data.code,
14       reference: data.reference,
15       designation: data.designation,
16       designation_uri: data.designation_uri,
17       cataclass: data.cataclass
18     }
19   });
20 },
21 delete: async (prisma: PrismaClient, code: number) : Promise<{ code: number; reference: string... => {
22   return prisma.cf1.delete({
23     where: {
24       code: code
25     }
26   })
27 }
28
29 };
```

Le service ce charge des appel en base de donnée , ici on utilise prisma , mais il est possible de faire du sql ou autres directement .

```
getAdrf: async (cusr: number, prisma: PrismaClient) : Promise<any> => {
  try {
    return await prisma.$queryRaw<any[]>`
      select * from lk_usr_adrf
      inner join adr on adr.code = lk_usr_adrf.cadr
      where cusr = ${cusr};
    `;
  } catch(err) {
    console.log(err)
  }
},
```

on crée ce fichier dans "src/services"

pour faire une méthode , il faut suivre une syntaxe claire :

[nom de la méthode] : async ([les paramètres]) => {[La méthode]}

Donc une fois que tout ceci est fait , la route devrais être accessible .

Le process est le suivant :

Route->Router->Controlleur->Service->Appel db

The screenshot shows a Postman interface with the following details:

- Request:** POST to `http://localhost:8000/api/auth/login`
- Headers:**

Key	Value	Description
Postman-Token	<calculated when request is sent>	
Content-Type	application/json	
Content-Length	<calculated when request is sent>	
Host	<calculated when request is sent>	
User-Agent	PostmanRuntime/7.52.0	
Accept	*/*	
Accept-Encoding	gzip, deflate, br	
Connection	keep-alive	
x-compte-token	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJjb2RlIjozNzY1LCJyY2xpIjo5ODEsImVtYWlsIjoic3VwcG9ydCszNzY1QGZzd3Byby5jb20iLCJpZGVudGlmYWVudCI6ImI0aG9tYXNAeXhpYS5mciIsImk0Mj9tcHRlIjoxLCJpYXQiOjE3NzU3MjM4MzIsImV4cCI6MTc3NTc1MjYzLn0.JVdBmnHgDHbpeZYLwSfrcsfpo5s_-jwFdainIIJUhyw	
- Response:** 200 OK, 876 ms, 567 B. The body is a JSON object:

```

1 {
2   "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJjb2RlIjozNzY1LCJyY2xpIjo5ODEsImVtYWlsIjoic3VwcG9ydCszNzY1QGZzd3Byby5jb20iLCJpZGVudGlmYWVudCI6ImI0aG9tYXNAeXhpYS5mciIsImk0Mj9tcHRlIjoxLCJpYXQiOjE3NzU3MjM4MzIsImV4cCI6MTc3NTc1MjYzLn0.JVdBmnHgDHbpeZYLwSfrcsfpo5s_-jwFdainIIJUhyw"
3 }

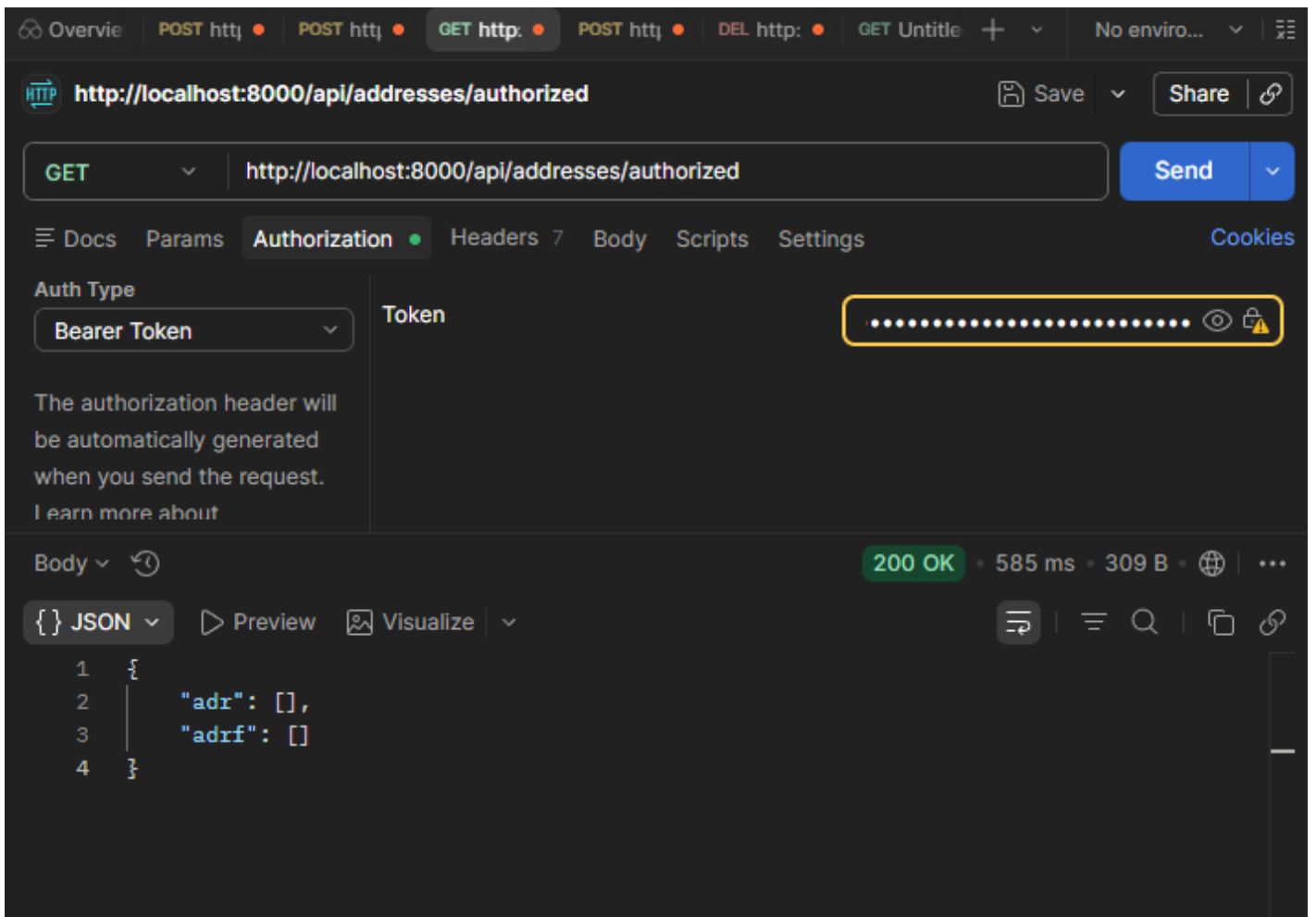
```

Aller sur la route "http://localhost:8000/api/auth/login"

passer les identifiants d'un utilisateur sur la db , en metant le token copié precedement dans un header "x-compte-token" : "token copié"

ca generera un autre token qu'il faudra copier a son tour et coller dans le champs

"bearer token" de l'auth type postman.



cette manipulation est a faire une fois , puis il faut simplement copié le token utilisateur généré au login dans chaque requete sur une route protégée comme on a vu précédemment.