

Node.js

C&DI – 2023–2024

Le Terminal

Afin de bien commencer le cours de Node.js, il est important de bien savoir naviguer dans son ordinateur.

Vous avez l'habitude de vous déplacer à travers les dossiers et fichiers depuis l'explorateur de fichiers Windows ou le Finder de Mac.

Mais, dans ce monde technique, on va prendre l'habitude de se déplacer depuis le terminal de commandes.

Ouvrez l'application « **Terminal** » sur Mac

Ouvrez l'application « **Invite de commandes** » sur Windows

Quelques commandes utiles :

ls pour macOS & linux / **dir** pour windows : permet de lister les dossiers et fichiers depuis l'endroit où on se trouve.

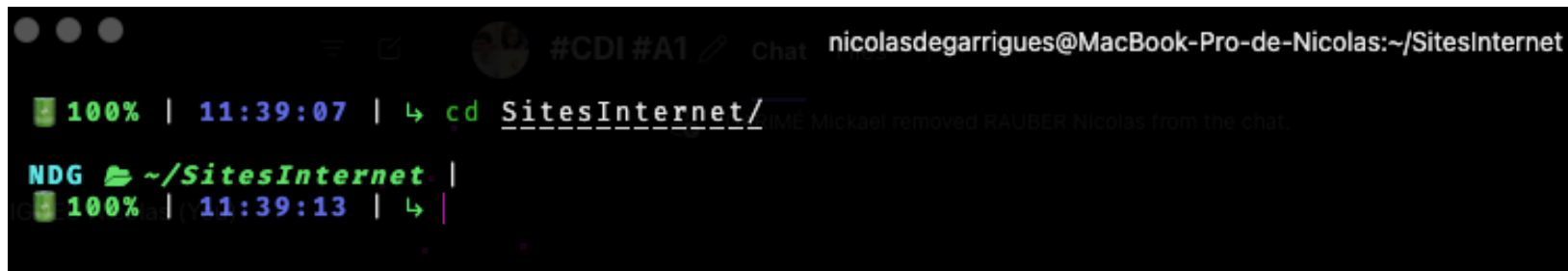
cd nomDuDossier : permet de se rendre dans le dossier par rapport à l'endroit où l'on se trouve. Pour éviter d'écrire le nom du dossier en entier, il est possible d'écrire seulement le début et de compléter en appuyant sur TAB

cd.. : permet de revenir au dossier parent par rapport à celui où l'on se trouve.

mkdir : permet de créer un dossier à l'endroit où l'on se trouve.

touch pour macOS & linux : permet de créer un fichier à l'endroit où on se trouve.

À côté du nom de mon ordinateur se trouve le dossier dans lequel je suis actuellement.



The screenshot shows a terminal window with a dark background. At the top, there's a title bar with window controls and a chat window titled "#CDI #A1" with a "Chat" button. The terminal prompt is "nicolasdegarrigues@MacBook-Pro-de-Nicolas:~/SitesInternet". The first line of the terminal shows a green battery icon at 100%, a timestamp of 11:39:07, and a green prompt character followed by "cd SitesInternet/". The second line shows the user "NDG" with a folder icon, the current directory "~/SitesInternet", and a green prompt character. The third line shows a green battery icon at 100%, a timestamp of 11:39:13, and a green prompt character followed by a vertical bar.

```
nicolasdegarrigues@MacBook-Pro-de-Nicolas:~/SitesInternet
100% | 11:39:07 | ↵ cd SitesInternet/
NDG ~/SitesInternet |
100% | 11:39:13 | ↵ |
```

Node.js, NPM...

Node.js est un environnement de développement open-source qui permet d'exécuter du code JavaScript côté serveur.

En d'autres termes, il permet aux développeurs de créer des applications web et des serveurs en utilisant JavaScript.

Cela permet également aux développeurs de ne se concentrer que sur un seul langage de programmation !

PHP JS LOVER ! <3



NPM ou Node Package Manager, est un outil qui permet de gérer des bibliothèques (majoritairement des morceaux de code pré-faits par d'autres utilisateurs pour simplifier les tâches chronophages).

NPM peut faire plusieurs choses :

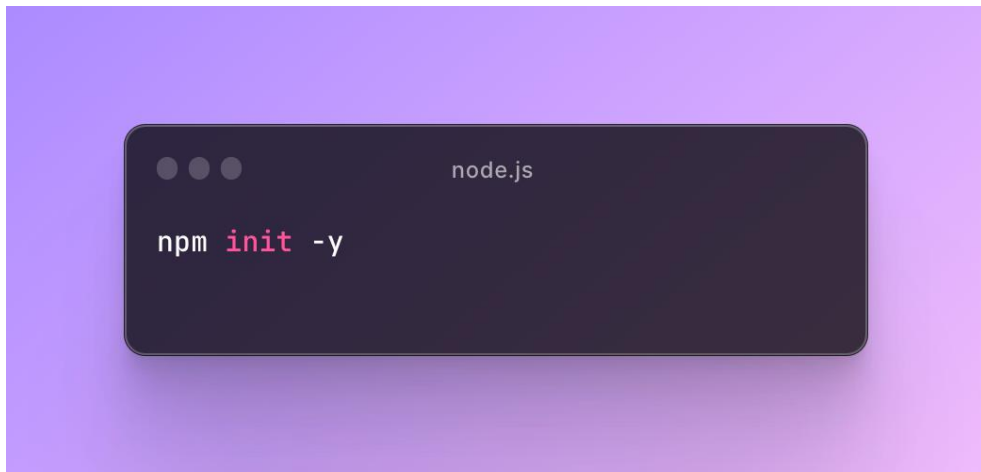
- Installation de paquets
- La gestion des dépendances
- Des scripts personnalisés (pour lancer le serveur par exemple)
- Des mises à jour faciles

<https://www.npmjs.com/>



Première commande Node.js

Pour initialiser un projet Node.js, il faut utiliser la commande :



Cela va créer un nouveau fichier appeler « **package.json** » à la racine du projet.

Le **package.json** est un fichier regroupant les informations liées au projet.

On y retrouve par exemple le nom du projet, sa version, ses dépendances associées...

Le format de ce fichier est un **JSON**.

Pour rappel, le JSON est un format qui permet d'écrire des informations de manière simple et légère.

```
{
  "name": "create-ppt",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Après avoir créé un fichier `index.js` à la racine, on va personnaliser notre `package.json`.

Dans un premier temps, nous allons rajouter la ligne qui va spécifier la syntaxe utilisée pour le projet.


Il existe 2 types de syntaxe: **ECMAScript Modules** (ESM) et **CommonJS** (CJS).

Nous allons utiliser ESM car la syntaxe est plus moderne, plus performante et propose plus de fonctionnalités.

De retour dans le index.js, on va simplement faire un :

```
console.log('Hello World !')
```

Puis on va lancer la commande **node** pour lire le fichier :

A terminal window with a dark background and a light purple border. It features three small grey circles in the top-left corner, representing window control buttons. The text 'node index.js' is displayed in a light grey monospace font.

```
node index.js
```

Notre **console.log()** s'affiche bien dans le terminal, on peut démarrer !

Express.js

Express.js est un **framework** pour **Node.js**.

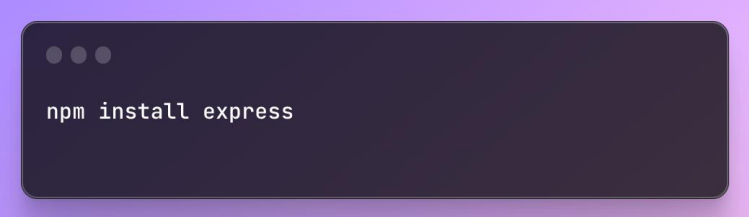
Il simplifie la création d'applications web en fournissant une structure et des fonctionnalités préconçues. Parmi ses différentes fonctionnalités, on retrouve :

- La gestion des routes
- L'intégration avec d'autres technologies
- Les middlewares
- ...

The logo for Express.js, featuring the word "express" in a lowercase, thin, sans-serif font.

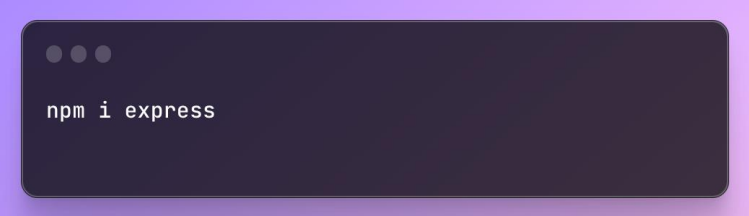
On va les découvrir au fur et à mesure de ce cours 😊

Pour installer une dépendance dans votre projet Node.js, il suffit d'utiliser **npm** :

A terminal window with a dark background and three light gray window control buttons in the top-left corner. The text 'npm install express' is displayed in a light gray monospace font.

```
npm install express
```

Au lieu d'écrire le mot **install**, les développeurs flemmards ont raccourci avec :

A terminal window with a dark background and three light gray window control buttons in the top-left corner. The text 'npm i express' is displayed in a light gray monospace font.

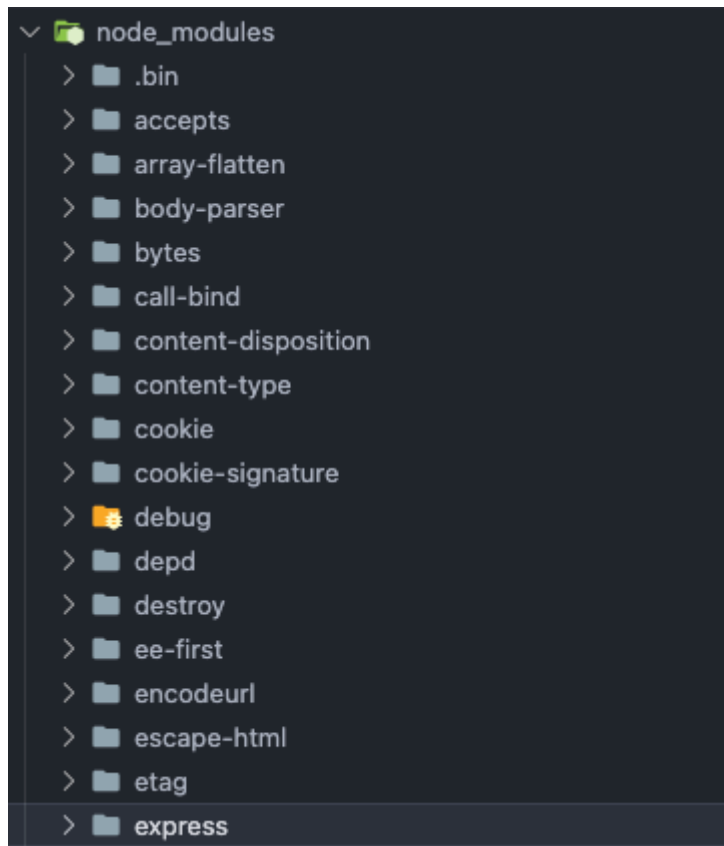
```
npm i express
```


Cette installation a créé un dossier **node_modules**, avec un grand nombre de sous-dossiers et fichiers.

Il contient les dépendances du projet, c'est-à-dire les bibliothèques et les modules externes dont l'application a besoin pour fonctionner correctement.

Le dossier doit être ignoré lors de la publication sur GitHub.

Et on touche seulement avec les yeux !



Création du serveur

On va maintenant procéder à la création de notre premier serveur.

On va tout d'abord importer **Express** que nous avons installé précédemment dans notre **index.js**.

Pour ça il suffit de faire en haut du fichier :

```
import express from 'express'
```

Nous allons ensuite initialiser notre application Express avec :

```
const app = express()
```

Maintenant que notre application est initialisée, on va faire tourner le serveur !

On va utiliser la fonction **listen** qui va écouter notre application sur le port 3000 :

```
app.listen(3000, () => {  
  console.log('Server is listening on port 3000')  
})
```

Pour lancer le serveur on va faire :



Si vous avez votre **console.log()** qui apparait dans votre terminal, c'est good !

Maintenant on va jouer avec la flemme du développeur !

Imaginons que nous ne voulons plus être sur le port 3000 mais 4000.

On le modifie dans le code mais cela ne prend pas directement effet, il faudrait couper le serveur et le redémarrer.

Heureusement il existe des solutions pour contrer ce potentiel problème de flemme.

On va transformer notre commande en rajoutant **--watch** qui va permettre de prendre en compte les modifications en temps réel :

A terminal window with a dark background and light text. The title bar at the top says "Node.js". The command "node --watch index.js" is entered in the terminal.

```
Node.js  
node --watch index.js
```

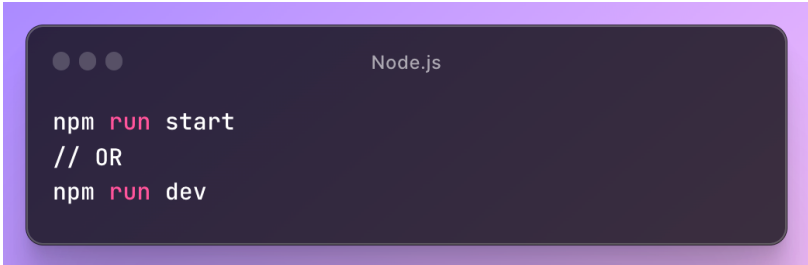
Toujours plus de flemmardise, on n'a pas envie de se rappeler les commandes à lancer.

Pour ça on peut les écrire dans le **package.json** pour les retenir et même les simplifier !

Pour lancer les différentes commandes désormais, on peut simplement écrire :

Parfait ! On a optimisé notre flemme !

```
"scripts": {  
  "start": "node index.js",  
  "dev": "node --watch index.js"  
},
```

A terminal window with a dark background and light text. The title bar says "Node.js". The content shows two lines of code: "npm run start" followed by "// OR" and then "npm run dev".

```
Node.js  
npm run start  
// OR  
npm run dev
```

Le Routing

En cours d'API, vous avez vu comment récupérer des données depuis une URL.

On va passer du côté obscur du code, et voir comment ça fonctionne dans un back-office !

Petit rappel des protocoles HTTP. Ce sont des requêtes guidées par une méthode :

- **GET** : récupérer des données
- **POST** : envoyer des données
- **PUT** : mettre à jour des données
- **DELETE** : supprimer des données

Pour créer une route avec Express, on va utiliser les fonctions nommées comme les méthodes **get()**, **post()**...

Dans ses paramètres, elle prend le **chemin** de la fonction qui lui sera associée, la **request** et le **result**.

À l'intérieur de **req**, on retrouve les éléments envoyés avec la requête (ex: les données d'un formulaire).

Le **res** va générer le résultat qui va être donné en échange de la requête.

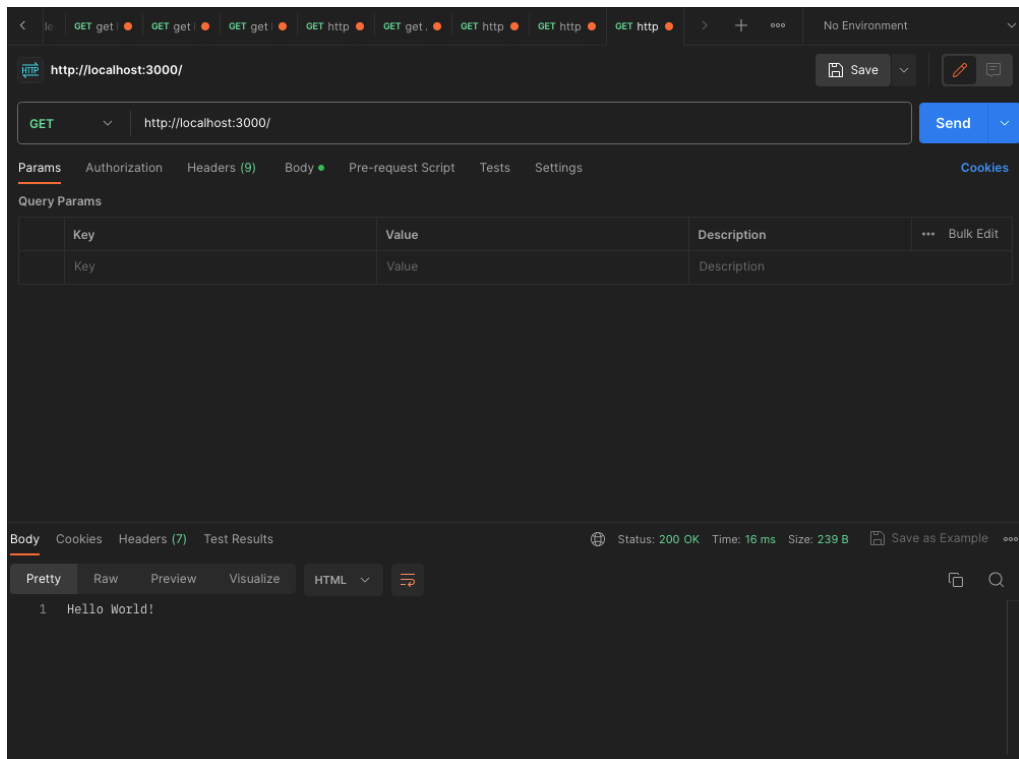
```
app.get('/', (req, res) => {  
  res.send('Hello World!')  
})
```

Ici on renvoie le message « Hello World! » avec la fonction **send**

Vous pouvez maintenant tester
votre nouvelle route disponible à
la racine du serveur sur Postman :

http://localhost:3000/

La route renvoie bien le message
Hello World!



À votre tour !

Vous allez faire la même chose pour une route **'/agents'** qui va lister les différents personnages de **Valorant** dans un tableau !

Pour information, il y a :

- | | | | |
|-------------|-----------|-----------|------------|
| - Jett | - Sage | - Skye | - Fade |
| - Raze | - Sova | - Yoru | - Harbor |
| - Breach | - Viper | - Astra | - Gekko |
| - Omen | - Cypher | - Kay/o | - Deadlock |
| - Brimstone | - Reyna | - Chamber | |
| - Phoenix | - Killjoy | - Neon | |

```
app.get('/agents', (req, res) => {  
  res.send([  
    'Jett',  
    'Raze',  
    'Breach',  
    '...'  
  ])  
})
```

The screenshot shows a web browser interface with a GET request to `http://localhost:3000/agents`. The response is a JSON array of agent names. The interface includes tabs for Params, Authorization, Headers (9), Body, Pre-request Script, Tests, and Settings. The Body tab is selected, showing the response in JSON format.

| Key | Value | Description |
|-----|-------|-------------|
| Key | Value | Description |

Body: Cookies Headers (7) Test Results Status: 200 OK Time: 15 ms Size: 420 B Save as Example

```
1  [  
2    "Jett",  
3    "Raze",  
4    "Breach",  
5    "Omen",  
6    "Brimstone",  
7    "Phoenix",  
8    "Sage",  
9    "Sova",  
10   "Viper",  
11   "Cypher",  
12   "Reyna",  
13   "Killjoy",  
14   "Skye",  
15   "Yoru",  
16   "Astra",  
17   "Kay/O",  
18   "Chamber",  
19   "Neon",  
20   "Fade",  
  ]
```

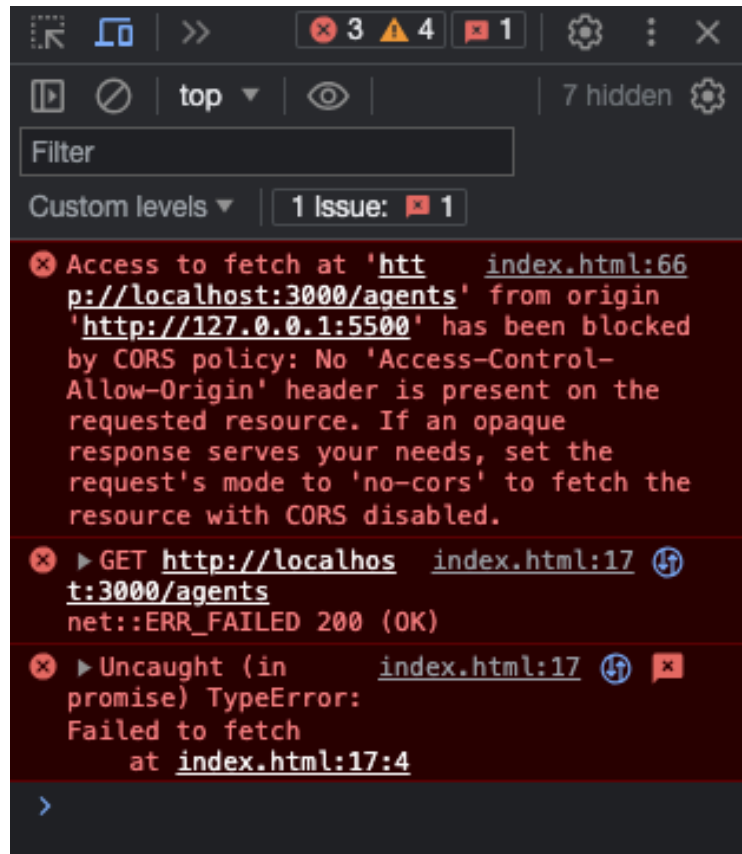
Affichage en front

Et malheureusement, cela ne va pas marcher !

Dans la console de votre navigateur, vous allez avoir une erreur de ce genre :

Il s'agit d'un problème de **CORS**.

Par défaut, le partage de ressources entre origines multiples est bloqué pour plus de sécurité



Pour débloquent ça, nous allons faire un :

 Node.js

```
npm i cors
```

Puis ajouter l'utilisation du package cors dans notre application avec :

```
import cors from 'cors'  
app.use(cors())
```


Une fois que les cors sont mis en place, notre liste d'agents apparaît !

List of agents

Jett

Raze

Breach

Omen

Brimstone

Phoenix

Sage

Sova

Viper

Cypher

Reyna

Killjoy

Skye

Yoru

Astra

Kay/o

Chamber

Neon

Fade

Architecture Node.js

Nous avons fait 2 routes, et ça prenait déjà pas mal de place dans notre fichier **index.js**.

Imaginez que vous ayez une vingtaine, une cinquantaine ou même une centaine de routes, cela deviendrait illisible !

On va donc séparer notre code en plusieurs fichiers pour une meilleure organisation !

Pour récupérer l'architecture : [Template Node](#)

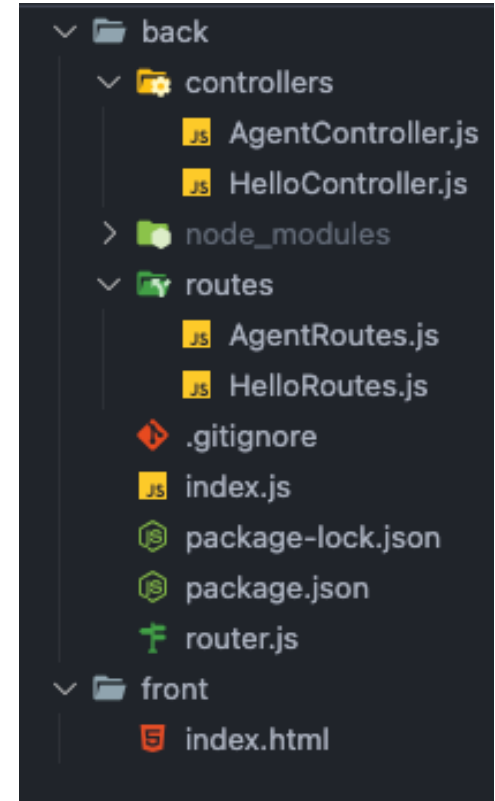
Vous remarquerez dans un premier temps que le front est désormais séparé du back.

On retrouve également de nouveaux dossiers dans la partie back :

- controllers
- routes

Et un nouveau fichier à la racine **router.js**.

On va voir ensemble comment on a séparé ça !



Dans notre **index.js**, on a importé notre **router.js** et utilisé dans notre application :

Dans le **router.js**, on a importé les routes liées aux Hello et Agents. Le chemin inscrit va être commun à toutes les routes incluses dans leur fichier respectif.

Puis on a exporté le router pour pouvoir l'importer dans l'**index.js** :

```
import router from './router.js'

app.use(router)
```

```
import express from 'express'
import agents from './routes/AgentRoutes.js'
import hello from './routes/HelloRoutes.js'

const router = express.Router()

router.use('/', hello)
router.use('/agents', agents)

export default router
```

Dans le dossier routes, on va retrouver les différentes routes séparées selon leurs utilités.

Par exemple dans le fichier **AgentRoutes.js**, on liste les différentes routes nécessaires aux Agents et au lieu d'écrire directement la fonction ici, on les place toutes dans un **controller**, et on appelle ses fonctions.

Puis on exporte pour pouvoir l'importer dans le **router.js**

```
import express from 'express'
import { getAgents, getAgent, createAgent, updateAgent, deleteAgent }
  from '../controllers/AgentController.js'

const router = express.Router()

router.get('/', getAgents)
router.get('/:id', getAgent)
router.post('/', createAgent)
router.put('/:id', updateAgent)
router.delete('/:id', deleteAgent)

export default router
```

Dans le dossier controllers, on va retrouver les différentes fonctions séparées selon leurs utilités.

Par exemple dans le fichier **AgentController.js**, on liste les différentes fonctions nécessaires aux Agents.

On y retrouve la fonction qu'on a faite auparavant.

Par la suite, on réalisera les autres fonctions.

```
const getAgents = (req, res) => {  
  res.send([  
    'Phoenix',  
    'Viper',  
    'Sage',  
    'Cypher',  
    'Reyna',  
    'Jett',  
  ])  
}  
  
const getAgent = (req, res) => {  
  //  
}  
  
...  
  
export {  
  getAgents,  
  getAgent,  
  ...  
}
```

Si on regarde dans le fichier **HelloController.js**, il y a un espacement pour réaliser la fonction **sayHello** qui retournera notre fameux « **Hello World!** »

En plus de ça, vous allez réaliser une nouvelle fonction qui s'appelle **sayHelloInFrench** et qui retournera « **Bonjour le Monde!** ».

Pour celle-ci, il faudra également créer la route **'/french'** et lier la fonction avec la route.

À vous de jouer !


```
// Ici on va écrire la fonction sayHello
const sayHello = (req, res) => {
  res.send('Hello World!')
}

// Fin de la fonction sayHello

const sayHelloInFrench = (req, res) => {
  res.send('Bonjour le Monde!')
}

export {
  sayHello,
  sayHelloInFrench,
}
```

```
import express from 'express'
import { sayHello, sayHelloInFrench } from '../controllers/HelloController.js'

const router = express.Router()

router.get('/', sayHello)
router.get('/french', sayHelloInFrench)

export default router
```

Prisma

Maintenant que l'on possède de bonnes bases en Node.js et Express.js, on va remplacer nos tableaux de data par de vraies informations stockées en base de données !

Il existe mille et une possibilités d'interagir avec une base de données en JS, mais Prisma possède quelques fonctionnalités plus sympathiques.

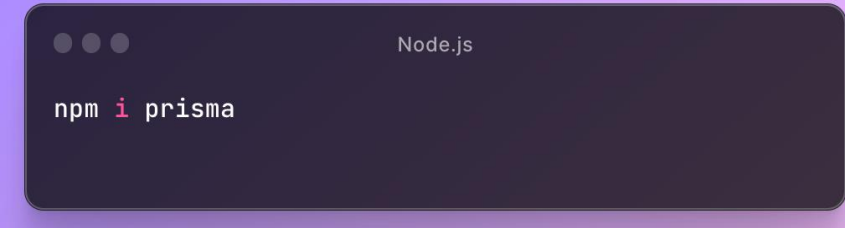
Prisma est ce qu'on appelle un ORM (Object-Relational Mapping). C'est un outil qui va faciliter l'interaction entre notre application et la base de données.

Imaginez cet outil magique qui va écrire du SQL à votre place !



Prisma

Pour installer Prisma sur notre projet :

A terminal window with a dark background and light text. The title bar shows three dots and the text "Node.js". The command "npm i prisma" is entered in the terminal.

```
Node.js  
npm i prisma
```

Pour ensuite l'initialiser dans notre projet, il faut faire la commande :

A terminal window with a dark background and light text. The title bar shows three dots and the text "Node.js". The command "npx prisma init" is entered in the terminal.

```
Node.js  
npx prisma init
```

Et on va se retrouver avec un nouveau fichier **prisma** et un nouveau fichier **.env**

On va procéder à la configuration, par défaut il utilise du PostgreSQL et nous allons utiliser MySQL.

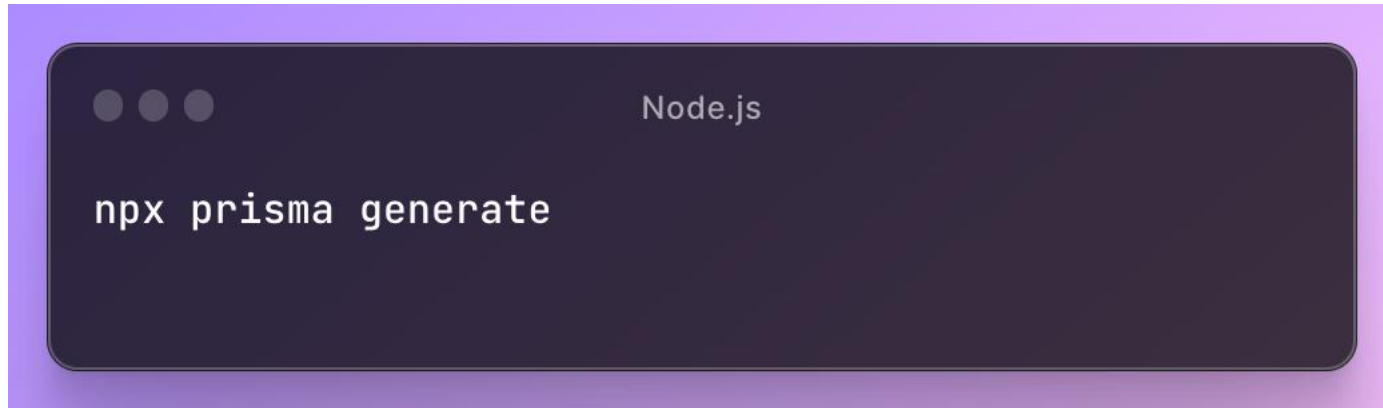
Dans le fichier **.env**, on va changer notre provider, ajouter nos identifiants et choisir le nom de notre base de données :

```
DATABASE_URL="mysql://root:root@localhost:3306/node-course?schema=public"
```

Dans le fichier **schema.prisma** présent dans le dossier **prisma**, nous allons également changer le provider pour **mysql** :

```
generator client {  
  provider = "prisma-client-js"  
}  
  
datasource db {  
  provider = "mysql"  
  url      = env("DATABASE_URL")  
}
```

Après avoir fait ses modifications, on va installer **prisma** et son **client** avec :

A terminal window with a dark purple background and rounded corners, set against a light purple background. The window has three small grey circles in the top-left corner and the text 'Node.js' in the top-right corner. The command 'npx prisma generate' is written in white text on the first line.

```
Node.js  
npx prisma generate
```

Imaginons que nous voulons une table **Agent** dans notre base de données, plus besoin de passer par phpMyAdmin !

Nous allons créer ce que l'on appelle un **model**, qui va contenir le squelette de la table.

Ici notre model Agent va avoir un **id**, un **name**, et un **createdAt** et **updatedAt**

```
model Agent {  
  id      Int      @id @default(autoincrement())  
  name    String  
  createdAt DateTime @default(now())  
  updatedAt DateTime @updatedAt  
}
```


Après avoir réalisé le model, on peut lancer une **migration**.

Une migration contient les modifications apportées à une structure de base de données.

La synchronisation entre la structure et la base de données se fera en même temps lors de cette commande :

Il faut ensuite lui donner un nom explicite.

Ici on va l'appeler **create_agent**

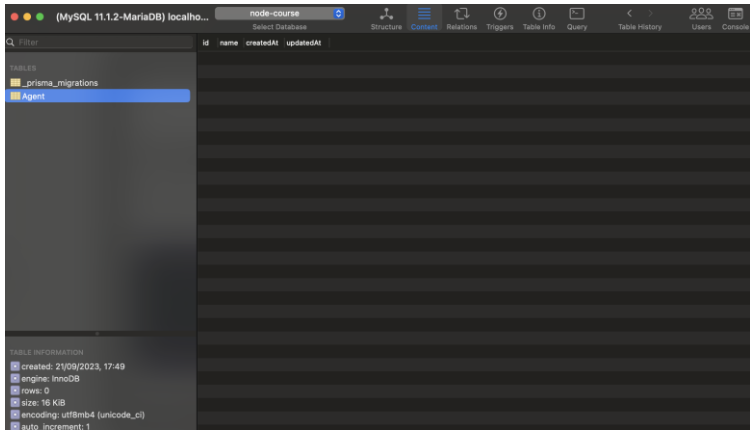
A terminal window with a dark background and light text. The title bar at the top says "Node.js". The command "npx prisma migrate dev" is entered in the terminal.

```
Node.js  
npx prisma migrate dev
```

Notre base de données avec la table Agent et un dossier de migration ont bien été créés !

Dans le dossier, on retrouve le fichier SQL qui a été généré automatiquement.

```
CREATE TABLE `Agent` (  
  `id` INTEGER NOT NULL AUTO_INCREMENT,  
  `name` VARCHAR(191) NOT NULL,  
  `createdAt` DATETIME(3) NOT NULL DEFAULT CURRENT_TIMESTAMP(3),  
  `updatedAt` DATETIME(3) NOT NULL,  
  
  PRIMARY KEY (`id`)  
) DEFAULT CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
```



Enregistrer en base de données

On va maintenant créer la fonction qui va stocker un nouvel agent en base de données.

Pour ça on va importer et initialiser le client de Prisma :

```
import { PrismaClient } from "@prisma/client"
const prisma = new PrismaClient()
```

On va maintenant remplir la fonction **createAgent**.

Comme précisé tout à l'heure, on va utiliser le **req** de la fonction pour récupérer les données qu'on va envoyer.

Les données sont stockées dans le **body** de la **request**.

Pour récupérer les informations, on va faire :

```
let agent = req.body
```

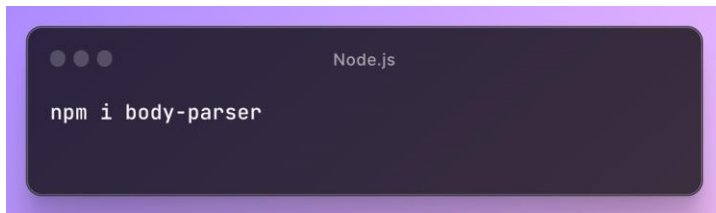
Pour tester si on récupère bien les informations on va faire un :

```
console.log(agent)
```

Et ça ne marche pas...

Il faut utiliser la dépendance **body-parser** qui va convertir le contenu envoyé en objet javascript exploitable.

Pour installer **body-parser** :

A terminal window with a dark background and light text. The title bar says "Node.js". The command "npm i body-parser" is entered in the terminal.

```
Node.js  
npm i body-parser
```

On va ensuite l'ajouter dans notre application :

```
import bodyParser from 'body-parser'  
  
app.use(bodyParser.json())  
  
app.use(bodyParser.urlencoded({ extended: true })))
```

Après avoir ajouté **body-parser**, on peut réessayer.
Et on récupère désormais les données envoyées !

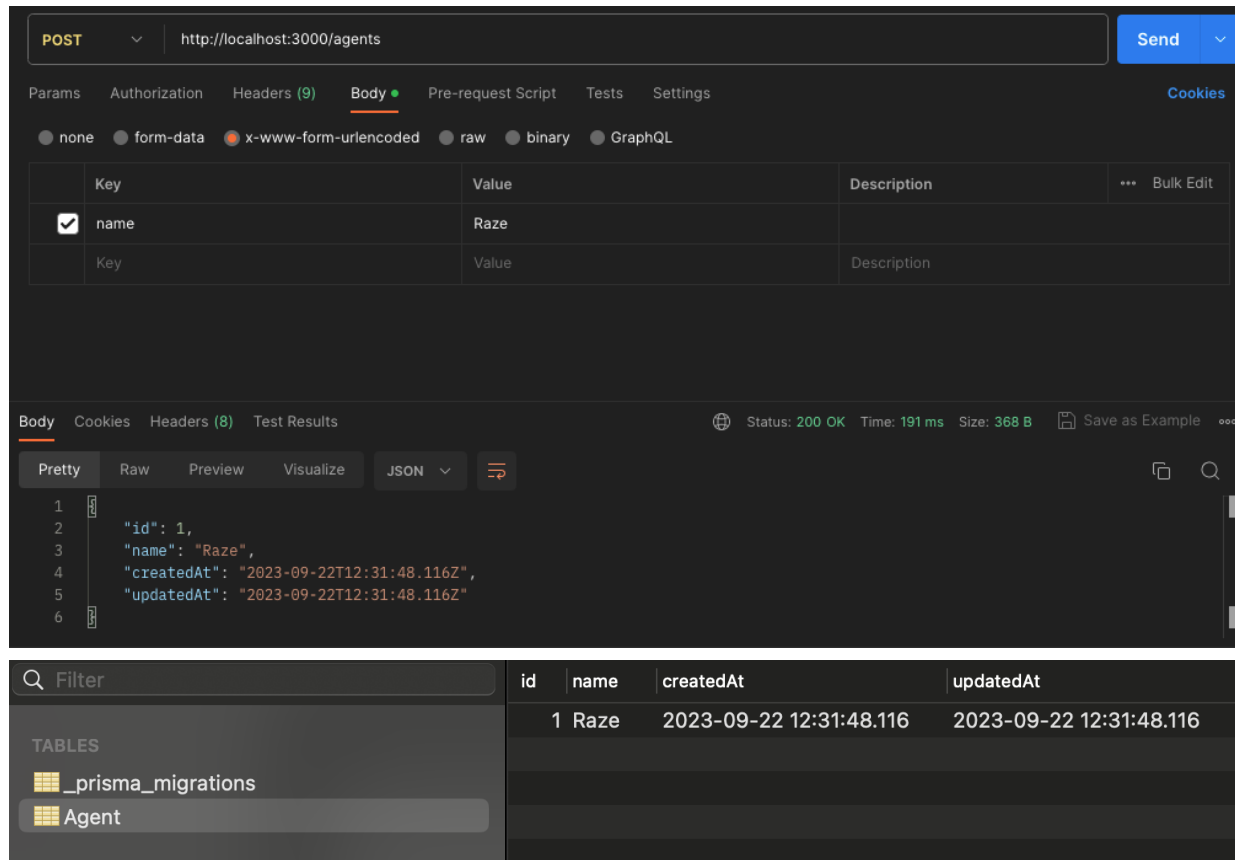
Pour enregistrer de la data dans la base de données, on va utiliser le **client** prisma, suivi du **model** concerné, puis la fonction **create()**.

On remplit ensuite le create avec les bonnes datas associées.

Selon le résultat de la requête, on envoie soit l'agent, soit une erreur.

```
prisma.agent
  .create({
    data: {
      name: agent.name,
    },
  })
  .then((agent) => {
    res.json(agent)
  })
  .catch((error) => {
    res.json(error)
  })
```

On peut tenter notre requête sur Postman, ça fonctionne !



POST http://localhost:3000/agents

Params Authorization Headers (9) **Body** Pre-request Script Tests Settings Cookies

none form-data **x-www-form-urlencoded** raw binary GraphQL

| Key | Value | Description |
|--|-------|-------------|
| <input checked="" type="checkbox"/> name | Raze | |
| Key | Value | Description |

Body Cookies Headers (8) Test Results

Status: 200 OK Time: 191 ms Size: 368 B Save as Example

Pretty Raw Preview Visualize JSON

```
1 {
2   "id": 1,
3   "name": "Raze",
4   "createdAt": "2023-09-22T12:31:48.116Z",
5   "updatedAt": "2023-09-22T12:31:48.116Z"
6 }
```

| id | name | createdAt | updatedAt |
|----|------|-------------------------|-------------------------|
| 1 | Raze | 2023-09-22 12:31:48.116 | 2023-09-22 12:31:48.116 |
| | | | |
| | | | |
| | | | |

Filter

TABLES

- _prisma_migrations
- Agent**

On va maintenant créer notre formulaire côté front pour ajouter notre agent.

```
<form>
  <input
    type="text"
    name="agent"
  />
  <button
    type="button"
    onclick="sendAgent()"
  >
    Send agent
  </button>
</form>
```

Puis envoyer les données du formulaire avec un **fetch()**.

Dans le fetch, on doit préciser la méthode POST ainsi que le type et le contenu que l'on va envoyer.

On redirige à la fin sur la même page.

```
const sendAgent = async () => {  
  let agent = document.querySelector('input[name="agent"]').value  
  
  let response = await fetch('http://localhost:3000/agents', {  
    method: 'POST',  
    headers: {  
      'Content-Type': 'application/json',  
    },  
    body: JSON.stringify({ name: agent }),  
  })  
  window.location.href = 'index.html'  
}
```

Afficher des données stockées en base

On va maintenant reprendre notre fonction **getAgents()** pour afficher nos agents stockés en base.

On va utiliser la fonction **findMany()** pour récupérer l'intégralité de nos agents.

```
prisma.agent
  .findMany()
  .then((agents) => {
    res.json(agents)
  })
  .catch((error) => {
    res.json(error)
  })
```

On va également adapter notre front pour bien préciser que nous voulons afficher le nom de l'Agent.

Et c'est tout !

```
<h2>${agent.name}</h2>
```

List of agents

Raze

Afficher les données spécifiques à un élément

Imaginons que l'on souhaite avoir une page dédiée à un agent, où on pourrait afficher ses compétences, son type...

La route est un peu spéciale puisqu'elle comprend un **binding**, c'est une information qui est variable.

```
router.get('/:id', getAgent)
```

Les informations affichées varieront selon ce qui est inscrit.

On va donc chercher l'id en paramètre avec :

```
let id = Number(req.params.id)
```

On va ensuite chercher l'agent qui possède l'ID correspondant.

Et c'est tout !

On va maintenant l'afficher sur une page spécifique.

```
prisma.agent
  .findUnique({
    where: {
      id: id,
    },
  })
  .then((agent) => {
    res.json(agent)
  })
  .catch((error) => {
    res.json(error)
  })
```


On va créer un fichier **agent.html**.

On va d'abord récupérer l'URL, puis récupérer le paramètre que l'on souhaite.

```
let url = window.location.search  
let id = new URLSearchParams(url).get('id')
```

Maintenant nous allons faire un **fetch** pour récupérer notre agent.

On va préciser l'agent que l'on souhaite avec le paramètre **id** dans l'URL.

On affiche ensuite les informations que l'on souhaite retrouvées sur la page, et un lien de retour.

On n'oublie pas de mettre la div agent dans l'HTML.

```
fetch(`http://localhost:3000/agents/${id}`)  
  .then((response) => response.json())  
  .then((data) => {  
    let agent = document.querySelector('#agent')  
    agent.innerHTML = `  
      <small>${data.id}</small>  
      <h1>${data.name}</h1>  
      <a href="index.html">Back</a>  
    `;  
  })
```

```
<div id="agent"></div>
```

Si on se rend sur notre page côté front et qu'on précise bien un id existant,

on tombe bien sur l'agent correspondant !

On va maintenant adapté notre page principale pour accéder directement à cette page.



Dans notre **index.html**, on va entourer le nom de l'agent avec un lien :

```
<a href="agent.html?id=${agent.id}">  
  <h2 ${agent.name}</h2>  
✂/a>
```

On place l'id en paramètre d'URL pour retrouver ensuite notre agent.

On peut maintenant naviguer entre les pages !

Modifier une entrée dans la base

Imaginons que nous avons fait une faute de frappe sur le nom du personnage. On va maintenant voir comment mettre à jour une donnée.

On va créer un fichier **rename.html**.

Dans ce fichier on va créer un formulaire pour le nom de l'agent.

Dans le bouton on précise que l'on va lancer la fonction **updateAgent()**.

On va la créer prochainement !

```
<form>
  <input
    type="text"
    name="agent"
  />
  <button
    type="button"
    onclick="updateAgent()"
  >
    Update agent
  </button>
</form>
```

Dans un premier temps on va récupérer le bon agent pour afficher le bon nom dans l'input.

On procède de la même manière que pour la page agent.

```
let url = window.location.search
let id = new URLSearchParams(url).get('id')

fetch(`http://localhost:3000/agents/${id}`)
  .then((response) => response.json())
  .then((data) => {
    let agent = document.querySelector('input[name="agent"]')
    agent.value = data.name
  })
```

Côté back-office, on va créer la fonction d'update.

On procède également de la même manière que pour la fonction **getAgent** sauf que l'on utilise la fonction update et on utilise les datas envoyées via le formulaire.

```
const updateAgent = (req, res) => {  
  let id = Number(req.params.id)  
  let agent = req.body  
  
  prisma.agent  
    .update({  
      where: {  
        id: id  
      },  
      data: {  
        name: agent.name  
      }  
    })  
    .then((agent) => {  
      res.json(agent)  
    })  
    .catch((error) => {  
      res.json(error)  
    })  
}
```


Retour côté front, on peut maintenant créer la fonction d'update.

On appelle l'url d'update, en précisant cette fois-ci la méthode **PUT**.

On a désormais notre formulaire de modification fonctionnel.

```
const updateAgent = async () => {  
  let agent = document.querySelector('input[name="agent"]').value  
  let response = await fetch(`http://localhost:3000/agents/${id}`, {  
    method: 'PUT',  
    headers: {  
      'Content-Type': 'application/json',  
    },  
    body: JSON.stringify({ name: agent }),  
  })  
  window.location.href = 'index.html'  
}
```

On rajoute ensuite un bouton sur **agent.html** pour lier les pages.

```
<a href="rename.html?id=${data.id}">Rename</a>
```

Supprimer une entrée de la base

On va maintenant supprimer une donnée de notre base.

On va dans un premier temps faire la fonction de suppression de l'agent en utilisant la fonction **delete**.

```
const deleteAgent = (req, res) => {  
  let id = Number(req.params.id)  
  
  prisma.agent  
    .delete({  
      where: {  
        id: id,  
      },  
    })  
    .then((agent) => {  
      res.json(agent)  
    })  
    .catch((error) => {  
      res.json(error)  
    })  
}
```

On va ensuite rajouter le bouton de suppression côté front et la fonction qui fait appel au back-office.

On utilise cette fois-ci la méthode DELETE.

Et c'est tout !

```
<button onclick="deleteAgent()">Delete</button>
```

```
const deleteAgent = async () => {  
  let response = await fetch(`http://localhost:3000/agents/${id}`, {  
    method: 'DELETE',  
  })  
  window.location.href = 'index.html'  
}
```

Inscription / Connexion

On va maintenant passer à la partie Inscription / Connexion.

Imaginons que l'on souhaite que notre utilisateur puisse se créer un compte et s'y connecter.

On va créer un nouveau Controller **AuthController.js**, un nouveau fichier de Routes **AuthRoutes.js** et on va les lier au router.

Ensuite, on va ajouter une nouvelle table **User** au schéma prisma.

```
model User {  
  id      Int      @id @default(autoincrement())  
  pseudo  String  
  email    String   @unique  
  password String  
  createdAt DateTime @default(now())  
  updatedAt DateTime @updatedAt  
}
```



A terminal window with a dark background and light text. The title bar shows three dots and the text 'Node.js'. The command 'npx prisma migrate dev' is entered in the terminal.

```
Node.js  
npx prisma migrate dev
```

On va également créer une page **signup.html**, dans lequel on va créer un formulaire avec **email, pseudo et password**.

On fait également un bouton pour lancer la fonction **signUp()**, que l'on va créer par la suite.

```
<form>
  <label for="email">Email</label>
  <input
    type="email"
    name="email"
  />
  <label for="pseudo">Pseudo</label>
  <input
    type="text"
    name="pseudo"
  />
  <label for="password">Password</label>
  <input
    type="password"
    name="password"
  />
  <button
    type="button"
    onclick="signUp()"
  >
    Sign Up
  </button>
</form>
```


Dans notre **AuthController.js**, on va créer une fonction **signUp()** qui va permettre à l'utilisateur de s'inscrire.

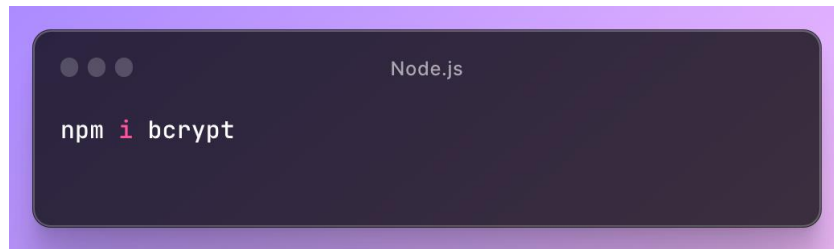
Même principe que pour les agents !

Enfin... pas tout à fait.

```
const signUp = (req, res) => {  
  const { email, pseudo, password } = req.body  
  prisma.user  
    .create({  
      data: {  
        email,  
        pseudo,  
        password,  
      },  
    })  
    .then((user) => {  
      res.status(200).json({ user })  
    })  
    .catch((error) => {  
      res.status(400).json({ error })  
    })  
}
```

Le mot de passe doit être hashé !
Pour ça on va utiliser l'outil **bcrypt**.

Pour installer **bcrypt** :

A terminal window with a dark background and light text. The title bar at the top says "Node.js". Inside the terminal, the command "npm i bcrypt" is entered. The letter "i" in "install" is highlighted in red.

```
Node.js  
npm i bcrypt
```

On import **bcrypt** en haut.

```
import bcrypt from 'bcrypt'
```

On utilise la fonction hash de **bcrypt**.

Il faut attendre la fin du hash pour passer à la suite, on va donc passer la fonction en asynchrone !

On n'oublie pas de mettre le **await** !

```
const signUp = async (req, res) => {  
  const { email, pseudo, password } = req.body  
  
  const hashedPassword = await bcrypt.hash(password, 10)  
  
  prisma.user  
    .create({  
      data: {  
        email,  
        pseudo,  
        password: hashedPassword,  
      },  
    })  
    .then((user) => {  
      res.json(user)  
    })  
    .catch((error) => {  
      res.json(error)  
    })  
}
```

Côté front, on va pouvoir faire notre fonction **signUp()** :

Notre inscription est finalisée.

```
const signUp = async () => {  
  let email = document.querySelector('input[name="email"]').value  
  let pseudo = document.querySelector('input[name="pseudo"]').value  
  let password = document.querySelector('input[name="password"]').value  
  
  let response = await fetch('http://localhost:3000/auth/signup', {  
    method: 'POST',  
    headers: {  
      'Content-Type': 'application/json',  
    },  
    body: JSON.stringify({  
      email: email,  
      pseudo: pseudo,  
      password: password,  
    }),  
  })  
  
  window.location.href = 'index.html'  
}
```

Pour la partie connexion, on va créer un fichier **login.html**.

Dans ce fichier, on va créer un formulaire, avec le bouton qui amène vers la fonction **login()**, que l'on va créer plus tard.

```
<form>
  <label for="email">Email</label>
  <input
    type="email"
    name="email"
  />
  <label for="password">Password</label>
  <input
    type="password"
    name="password"
  />
  <button
    type="button"
    onclick="login()"
  >
    Log In
  </button>
</form>
```

Dans le **AuthController.js**,
on va créer une fonction
login().

On récupère les
informations de l'utilisateur
dans le body puis on
requête à Prisma dans
notre BDD

```
const { email, password } = req.body

const user = await prisma.user.findUnique({
  where: {
    email,
  },
})
```

On va vérifier dans un premier qu'un utilisateur avec l'adresse email demandée existe.

Si il n'existe pas, on retourne une erreur 404.

```
if (!user) {  
  return res.status(404).json({ error: 'User not found' })  
}
```

Ensuite on va vérifier que le mot de passe correspond. Pour ça on va utiliser la fonction **compare()** de bcrypt.

On retourne une erreur si le mot de passe ne correspond pas.

Il ne nous reste qu'une seule chose à faire :
Créer un token
d'authentification !

```
const validPassword = await bcrypt.compare(password, user.password)

if (!validPassword) {
  return res.status(404).json({ error: 'Password not valid' })
}
```


Tokens d'authentification (JWT)

Pour qu'un utilisateur soit reconnu tout au long de son parcours sur notre site, il faut qu'il se « balade » avec une clé qui le représente. Cela s'appelle un **token**.

C'est comme si vous deviez présenter votre carte d'identité quand vous allez quelque part.

Ce système qui crée et vérifie nos tokens s'appelle **JSON Web Token** (JWT).

On va l'installer avec :

A terminal window with a dark background and a light purple border. It contains the command 'npm i jwt' where 'i' is highlighted in red. There are three small grey circles in the top left corner of the terminal window, representing window control buttons.

```
npm i jwt
```

Ensuite on va vérifier que le mot de passe correspond. Pour ça on va utiliser la fonction **compare()** de bcrypt.

On retourne une erreur si le mot de passe ne correspond pas.

Il ne nous reste qu'une seule chose à faire :
Créer un token
d'authentification !

```
const validPassword = await bcrypt.compare(password, user.password)

if (!validPassword) {
  return res.status(404).json({ error: 'Password not valid' })
}
```

Pour créer le token, on va demander à JWT de le **signer**.

On va préciser quelques informations utilisateurs rendre le JWT unique et plus sécuriser, en revanche on ne met pas d'informations sensibles (ex: mot de passe...) mais seulement 1 ou 2 informations qui représente bien notre utilisateur.

On va également passer un mot ou une phrase secrète propre à notre site, qui va être hashé avec le reste de nos informations pour plus de sécurité.

Enfin on précise le temps d'expiration de notre token. Il ne sera plus valable pour être authentifié après la date passée.

```
const token = jwt.sign({ id: user.id, email: user.email }, process.env.JWT_SECRET, {  
  expiresIn: '2h',  
})
```

On va pour finir retourner le token.

```
res.json(token)
```

On retourne côté front et on va pouvoir faire notre fonction de login.

```
let email = document.querySelector('input[name="email"]').value
let password = document.querySelector('input[name="password"]').value

let response = await fetch('http://localhost:3000/auth/login', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
  },
  body: JSON.stringify({
    email: email,
    password: password,
  }),
})
```

On va terminer en enregistrant notre token retourné dans le **localStorage**.

Le localStorage est un espace pour sauvegarder des informations dans notre navigateur.

On n'oublie pas de retourner vers l'index.html à la fin.

On peut désormais se connecter !

```
let data = await response.json()
localStorage.setItem('token', data)

window.location.href = 'index.html'
```

On va également ajouter notre système de token dans la partie signup pour que l'utilisateur soit directement connecté après l'inscription.

- Dans la partie **AuthController.js - signup()**

- Dans la partie **signup.html**

```
prisma.user
.create({
  data: {
    email,
    pseudo,
    password: hashedPassword,
  },
})

.then((user) => {
  const token = jwt.sign({ id: user.id, email: user.email }, process.env.JWT_SECRET, {
    expiresIn: '2h',
  })

  res.json(token)
})

.catch((error) => {
  res.json(error)
})
```

Page profil

On va créer une page profil.

Cette page profil ne peut être accessible que si l'utilisateur est authentifié. On le redirige vers la page de login.

Pour se faire on va créer un **profil.html**.

On va créer plus tard la fonction qui va requêter notre back.

```
<div id="user"></div>
```

Côté back, on va créer un **UserController.js**, un **UserRoutes.js** et les lier à au router avec `/user` comme préfix.

On va créer une fonction **getProfile()**, qui va d'abord récupérer, si il existe, le token que l'on va envoyer dans un endroit spécifique (le header).

Puis on va vérifier que le token correspond bien à la connexion attendue.

Après, on récupère le bon utilisateur.

```
const token = req.headers['x-access-token']

if (!token) {
  return res.status(401).json({ error: 'No token provided' })
}

jwt.verify(token, process.env.JWT_SECRET, (error, decoded) => {
  if (error) {
    return res.status(401).json({ error: 'Unauthorized' })
  }

  prisma.user
    .findUnique({
      where: {
        id: decoded.id,
      },
    })
    .then((user) => {
      res.json(user)
      console.log(user)
    })
    .catch((error) => {
      res.json(error)
    })
  })
})
```

Côté front, on récupère le token.

Si il n'existe pas, on redirige l'utilisateur vers la page de connexion.

Sinon on requête l'utilisateur en précisant notre token dans le header.

On affiche ensuite les informations de notre utilisateur.

```
let token = localStorage.getItem('token')

if (!token) {
  window.location.href = 'login.html'
}

fetch('http://localhost:3000/user/', {
  headers: {
    'x-access-token': `${token}`,
  },
})
  .then((response) => response.json())
  .then((data) => {
    let user = document.querySelector('#user')
    user.innerHTML = `
      <h1>${data.pseudo}</h1>
      <h2>${data.email}</h2>
      <a href="index.html">Back</a>
    `
  })
```

Le relationnel

Imaginons que chaque utilisateur peut ajouter sur son profil son agent favori.

On va devoir donc lier un agent à un utilisateur.

On va donc modifier notre schéma Prisma :


On ajoute un identifiant d'agent dans la table user et on précise les références.

```
model Agent {  
  id      Int      @id @default(autoincrement())  
  name    String  
  users   User[] // new  
  createdAt DateTime @default(now())  
  updatedAt DateTime @updatedAt  
}  
  
model User {  
  id      Int      @id @default(autoincrement())  
  pseudo  String  
  email   String   @unique  
  password String  
  agent   Agent?   @relation(fields: [agentId], references: [id]) // new  
  agentId Int? // new  
  createdAt DateTime @default(now())  
  updatedAt DateTime @updatedAt  
}
```

Pour appliquer les modifications :

on donne un nom logique à notre migration comme :

update_user_and_agent_tables

A terminal window with a dark background and light text, showing the command `npx prisma migrate dev`. The window has three small circles in the top left corner, representing window controls.

```
npx prisma migrate dev
```

On va rajouter un formulaire sur la page profil pour ajouter son Agent favori :

Ensuite on va requêter les agents pour les récupérer et les placer dans le select :

```
<form>
  <select></select>
  <button onclick="favoriteAgent()">Favori</button>
</form>
```

```
fetch('http://localhost:3000/agents')
  .then((response) => response.json())
  .then((data) => {
    let select = document.querySelector('select')
    data.forEach((agent) => {
      select.innerHTML += `
        <option value="${agent.id}">${agent.name}</option>
      `
    })
  })
```

Du côté back, on va créer la fonction d'ajout de l'agent en favori.

On récupère le token et vérifie si il est valide.

```
const token = req.headers['x-access-token']

if (!token) {
  return res.status(401).json({ error: 'No token provided' })
}

jwt.verify(token, process.env.JWT_SECRET, (error, decoded) => {
  if (error) {
    return res.status(401).json({ error: 'Unauthorized' })
  }
})
```


Si il est valide, on modifie la colonne d'agentId dans notre utilisateur :

```
prisma.user
  .update({
    where: {
      id: decoded.id,
    },
    data: {
      agentId: Number(req.body.agent),
    },
  })

  .then((user) => {
    res.json(user)
  })

  .catch((error) => {
    res.json(error)
  })
```

On va rapidement modifier notre requête pour chercher notre utilisateur.

On souhaite maintenant récupérer l'agent en lien avec l'id.

Il suffit simplement d'ajouter un **include** avec le nom du model en question :

```
prisma.user
  .findUnique({
    where: {
      id: decoded.id,
    },
    include: {
      agent: true,
    },
  })
  .then((user) => {
    res.json(user)
  })
  .catch((error) => {
    res.json(error)
  })
```

On va maintenant côté front, créer notre fonction pour envoyer notre nouvel agent favori.

On précise le token dans le header et l'id de l'agent dans le body.

```
const favoriteAgent = async () => {  
  let agent = document.querySelector('select').value  
  let response = await fetch('http://localhost:3000/user/favorite', {  
    method: 'POST',  
    headers: {  
      'Content-Type': 'application/json',  
      'x-access-token': `${token}`,  
    },  
    body: JSON.stringify({ agent: agent }),  
  })  
  window.location.href = 'profile.html'  
}
```

On va également afficher l'agent favori de l'utilisateur si il en a un.

Si il n'en a pas, on affiche « pas d'agent favori ».

```
<p>${data.agent?.name ?? "pas d'agent favori"}</p>
```