POLYTECH DIJON

TRAVAUX DIRIGÉS

NodeJS – TypeScript

Créer une API

OBJECTIFS

- Développer une API web faisant l'interface entre une application cliente et une base de données.
- Découvrir l'environnement NodeJS
- Découvrir le langage TypeScript

PREPARATION

PREREQUIS

- NodeJS doit être installé.

PROJET

- Créez un dossier pour le projet du TD.
- Ouvrez ce dossier dans VS Code.

FRONTEND

- Créez un sous-dossier nommé `front` et qui contiendra les fichiers de l'archive suivante : https://www.lamarmotte.info/wp-content/uploads/2025/05/front.zip

- Sélectionnez le fichier `index.html` présent dans le dossier et démarrez Live Server.
- Vérifiez que le navigateur affiche quelque chose de semblable à la capture ci-dessous.

charles.meunier@ube.fr

BACKEND

- Créez un sous-dossier nommé `api` dans le dossier du projet (le premier que vous avez créé).
- Créez un sous dossier `src` dans le dossier `api`.

TYPESCRIPT

> **Important**
> Pour la suite du sujet vous veillerez à vous placer dans le dossier `api` lors de l'installation de paquets ou pour exécuter la commande `tsc`.

INSTALLATION

- Installez TypeScript comme vous vu dans le cours.

CONFIGURATION

Une fois TypeScript installé, il est nécessaire de le configurer pour votre projet.

- Exécutez la commande suivante pour créer le fichier de configuration TypeScript par défaut :

```
npx tsc --init
```

- Editez le fichier `tsconfig.json` qui vient d'être créé dans le dossier `api`. Ce fichier propose tout un ensemble de paramètres qui peuvent être appliqués à l'environnement TypeScript.
- Recherchez le paramètre `rootDir` et configurez le avec la valeur `./src`. De cette façon, TypeScript cherchera les fichiers sources de votre projet dans le dossier `src`.
- Recherchez le paramètre `outDir` et configurez le avec la valeur `./dist`. Ceci permettra de ne pas mélanger les fichiers TypeScript avec les fichiers JavaScript qui seront générés à la transpilation. Ainsi, vous créerez vos fichiers TypeScript dans le dossier `src` et les fichiers JavaScript seront générés dans le dossier `dist` (le dossier sera créé automatiquement).
- Modifiez à `false` les paramètres `sourceMap`, `declaration`, `declarationMap` et `verbatimModuleSyntax`.

> **Note**
> Pensez à enregistrez le fichier `tsconfig.json` après modification et à décommentez les lignes que vous avez modifiées.

VERIFICATION

- Créez un fichier `index.ts` dans le dossier `src` de votre projet.
- Copiez le code suivant dans le fichier `index.ts` :

```
console.log("Hello World !");
```

- Transpilez votre projet avec la commande suivante :

```
npx tsc
```

- Vérifiez que le dossier **`dist`** a bien été créé dans le dossier api et qu'il contient un fichier **`index.js`**

EXECUTION

- Pour exécuter votre code, utilisez la commande suivante en vous plaçant préalablement dans le dossier **`api/dist`** :

```
node index.js
```

- Vous pouvez également utilisez la commande suivante :

```
node ./
```

Par défaut, NodeJS cherchera un fichier nommé **`index.js`** comme point d'entrée de votre programme.

PREMIER SERVEUR WEB

- Installez le paquet **`fastify`** qui fournit les éléments nécessaires à la création d'un service Web :

```
npm i fastify
```

> **Important**
> Vérifiez toujours dans quel dossier vous êtes avant d'installer un paquet. Sinon, il sera placé dans le mauvais dossier. Ici, vous devez être dans le dossier **`api`**.

> **Note**
> Ici nous ne précisions pas l'option **`-D`** car ce paquet sera nécessaire au fonctionnement de l'application finale, en production.

- Remplacez le code du fichier **`index.ts`** par celui-ci :

```
//Importe la fonction Fastify depuis le paquet fastify
import Fastify from "fastify";

//Crée une instance de Fastify
const fastify = Fastify();

//Crée une route HTTP GET vers l'URL /
fastify.get("/", (request, reply) => {
    //Envoie une string en réponse au client
    reply.send("Hello World !");
});

//Démarre le serveur Web qui écoutera sur le port 8080
fastify.listen({port : 8080});
```

Important
Ne modifiez pas le fichier **`index.js`**. Son contenu sera regénéré automatiquement par TypeScript à la prochaine transpilation.

- Transpilez votre code TypeScript.
- Exécutez votre programme.
- Testez le bon fonctionnement en accédant à l'adresse http://localhost:8080 à l'aide de votre navigateur.

Astuce
Pour éviter d'avoir à exécuter la commande **`tsc`** à chaque modification, vous pouvez exécuter le compilateur TypeScript en mode Watch. De cette façon, il transpilera votre code à chaque fois que celui-ci sera modifié :

```
npx tsc -w
```

BASE DE DONNEES

PREMIERE REQUETE

- Créez un sous-dossier **`database`** dans le dossier **`api`** et placez y le fichier **`db.sqlite`** présent dans l'archive suivante :
  https://www.lamarmotte.info/wp-content/uploads/2025/05/db.zip

  Le modèle relationnel de la base de données est présenté en annexe 1.

- Installez le paquet **`better-sqlite3`** (nécessaire au fonctionnement de l'application pour dialoguer avec la base de données).
- Installez le paquet **`@types/better-sqlite3`** (nécessaire **uniquement durant la phase de développement** et permettant à TypeScript d'identifier les types de chaque élément proposé par le paquet **`better-sqlite3`**).

- A partir de la documentation disponible en annexe 2, créez une route `GET /users` qui retourne la liste des utilisateurs présents dans la base de données.
- Testez le bon fonctionnement en accédant à l'adresse http://localhost:8080/users à l'aide de votre navigateur.

CORS POLICY

- Affichez de nouveau la page du frontend et observez la console du navigateur.
  Vous devriez obtenir une erreur ressemblant à :

```
Access to fetch at 'http://localhost:8080/users' from origin '    localhost/:1
http://localhost:5173' has been blocked by CORS policy: No 'Access-Control-
Allow-Origin' header is present on the requested resource.
```

- Expliquez ce qu'est le CORS.

Fastify propose un module qui configure automatiquement le CORS du serveur pour accepter les requêtes provenant d'autres domaines. Il est également possible de mettre en place une configuration plus restrictive.

- Installez le module `@fastify/cors`.
- Utilisez les informations suivantes pour activer le CORS sur votre serveur web :

```
//Importe le module CORS de Fastify
import cors from "@fastify/cors";

//Ajoute le module CORS à l'instance du serveur Web
fastify.register(cors);
```

- Vérifiez la page du frontend (cliquez sur le bouton « actualiser » si besoin).

DEVELOPPEMENT DE L'API

Le front de l'application Lovytech a été conçu en amont de l'API mais en respectant la documentation disponible en annexe 3.

LISTE DES UTILISATEURS

Ici, plus rien à faire. Next !

CREATION DE COMPTE

Lorsque vous cliquez sur le nom d'un utilisateur pour entamer une conversation, vous êtes invité à vous connecter ou à créer un compte.

- Créez une nouvelle route `/users` mais qui cette fois-ci utilisera la méthode `POST`.
- Etudiez la documentation de l'API en annexe 3.2.

  L'API `POST /users` reçoit les informations du formulaire d'inscription (name, mail et password) et les utilise pour créer un utilisateur dans la base de données.

Aucun des trois champs ne doit être vide et le mail doit être unique pour chaque utilisateur.

Le serveur retournera un code 201 si le compte a pu être créé, 409 si un compte avec cette adresse e-mail existe déjà ou 500 si une erreur survient durant le traitement.

- Codez la fonction attachée à la nouvelle route.

Quelques éléments utiles

```
// Récupérer des données transmises dans le body de la requête
// ReceiveDataType est un type personnalisé indiquant à TypeScript
// ce qu'est censé contenir request.body (cela veut dire que vous
devez remplacer ReceiveDataType par le nom du type que vous aurez
créé vous)
const { data1, data2 } = request.body as ReceiveDataType;

// Insérer des données dans la base
// Attention : les données viennent de l'extérieur. On utilisera
// donc une requête paramétrique pour éviter les injections SQL.
const query = db.prepare("INSERT INTO …");
const result = query.run(param1, param2);

// Indique le nombre de lignes affectées par la requête
result.changes

// Dernier identifiant créé par un auto-incrément
result.lastInsertRowid

// Envoyer des données au format JSON
reply.send(data);

//Envoyer un code de retour sans donnée
reply.code(999).send();
```

Note
Pour le moment, vous stockerez les mots de passe en clair dans la base de données. Oui, c'est mal. Et oui, le hachage des mots de passe est expliqué un peu plus loin.

- Testez le bon fonctionnement et vérifiez la présence du ou des comptes créés dans la base de données.

CONNEXION

Maintenant que le compte de l'utilisateur est créé, il faut qu'il puisse se connecter à l'application.

- Créez une route `/auth` qui utilise la méthode POST.
- Etudiez la documentation de l'API en annexe 3.3.

L'API `POST /auth` reçoit les informations du formulaire de connexion (mail et password) et les utilise pour trouver l'utilisateur correspondant et générer le token de sécurité.

Le serveur retournera un code 200 accompagné du token si les identifiants fournis correspondent à un utilisateur. Sinon, un code 404 sera retourné en cas de non-correspondance. Enfin, en cas d'erreur durant le processus, un code 500 sera envoyé au client.

L'annexe 4 fournit des informations sur la génération de Json Web Token (JWT).

- Codez la fonction attachée à la route de connexion.
- Testez le bon fonctionnement.

ENVOI DE MESSAGES

Ça y est, vous êtes connecté ! L'amour est à votre portée !

- Créez une nouvelle route /messages qui utilisera la méthode `POST`.
- Etudiez la documentation de l'API en annexe 3.4.
  L'API `POST /messages` reçoit l'identifiant du destinataire ainsi que le contenu du message.

  Attention ! La requête ne doit être traitée que si elle est pourvue d'un entête `Authorization` contenant un token valide.

  L'identifiant du destinataire du message est dans le body de la requête et l'identifiant de l'émetteur du message est dans le payload du token.

  Le serveur renverra un code 200 accompagné des informations du message dont l'id en base de données du message enregistré. Si le token de sécurité est invalide, le serveur renverra un code 401. Si l'utilisateur destinataire est inconnu, le serveur répondra avec un code 404. Enfin, si une erreur se produit, le serveur renverra un code 500.

- Codez la fonction attachée à la route.
- Testez le bon fonctionnement et vérifiez l'ajout des messages à la base de données.

CHARGEMENT DES MESSAGES

Il manque une dernière route permettant de charger les messages d'une discussion.

- Créez une route `/messages` qui utilise la méthode `GET`.
- Etudiez la documentation de l'API en annexe 3.5.

  L'API `GET /messages` reçoit l'identifiant de la personne avec qui l'utilisateur a échangé.

  Attention ! La requête ne doit être traitée que si elle est pourvue d'un entête `Authorization` contenant un token valide.

  Le serveur renverra un code 200 accompagné d'un tableau de messages correspondant aux éléments de la discussion. Si le token de sécurité est invalide, le serveur renverra un code 401. En cas d'erreur, un code 500 sera retourné.

- Codez la fonction attachée à la route.
- Testez le bon fonctionnement en actualisant la page des conversations.

SECURITE

Evidemment, stocker les mots de passe en clair dans une base de données n'est pas une bonne idée. Pis encore, c'est criminel. Sérieusement. En tant que professionnels de l'informatique vous engagez votre responsabilité en ne respectant pas ce genre de règles.

Heureusement, JavaScript est aujourd'hui doté d'outils de cryptographie très sérieux permettant, entre autres, de hacher des informations.

- A partir des informations fournies en annexe 5, modifiez les fonctions de création de compte et de connexion pour travailler avec des mots de passe hachés.
- Testez le bon fonctionnement.

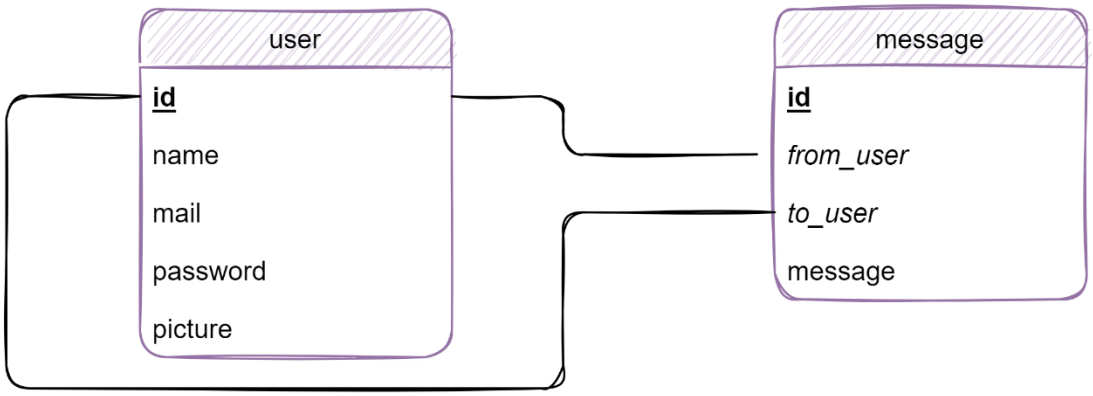
BONUS

Actuellement, si deux utilisateurs ont le même mot de passe, ils auront le même hash mémorisé dans la base de données. En cas de fuite de cette dernière, et si le nombre d'utilisateurs inscrits est assez grand, les hash les plus nombreux risquent, statistiquement, de correspondre aux mots de passe les plus utilisés donnant aux Hackers des indices sur les comptes les plus vulnérables.

- Proposez et implémentez une solution pour rendre chaque hash unique

ANNEXES

ANNEXE 1 – MODELE RELATIONNEL DE LA BASE DE DONNEES LOVYTECH



ANNEXE 2 – DOCUMENTATION DE BETTERSQLITE3

- Importer le module BetterSqlite3

```
import Sqlite3 from "better-sqlite3";
```

- Ouvrir une connexion à une base de données SQLite

```
const db = Sqlite3("pth/to/db.sqlite");
```

- Préparer une requête SQL

```
const query = db.prepare("SELECT * FROM table;");
const results = query.all();
```

- Préparer une requête SQL paramétrique

```
const query = db.prepare("SELECT * FROM table WHERE field = ?;");
const results = query.all("field_value");
```

Toute la documentation de BetterSqlite3.

ANNEXE 3 – DOCUMENTATION DE L'API LOVYTECH

3.1 - Récupérer la liste des utilisateurs inscrits

Client request	
URL	/users
Method	GET
Server response	
Content-Type	application/json
Body	[{ id : number, name : string, picture : string }, …]
Response code	200 – OK 500 – Internal serveur error

3.2 - Enregistrement d'un utilisateur

Client request	
URL	/users
Method	POST
Content-Type	application/json

Body	{
	name : string,
	mail : string,
	password : string
	}

Server response	
Response Code	204 – OK : account created 409 – Conflict : account already exists 500 – Internal server error

3.3 - Connexion d'un utilisateur

Client request	
URL	/auth
Method	POST
Content-Type	application/json

Body	{
	mail : string,
	password : string
	}

Server response	
Content-Type	text/plain
Body	"security_token"
Response Code	200 – OK : connection successful 404 – Not found 500 – Internal server error

3.4 - Envoyer un message

Client request	
URL	`/messages`
Method	`POST`
Authorization	`Bearer security_token`
Content-Type	`application/json`

Body	`{`
	`  to_user : number,`
	`  message : string`
	`}`

Server response	
Content-Type	`application/json`

Body	`{`
	`  id : number,`
	`  from_user : number,`
	`  to_user : number,`
	`  message : string`
	`}`

Response Code	`200 – OK : message registered`
	`401 – Unauthorized : invalid security token`
	`404 – Not found : can’t find « to » user`
	`500 – Internal server error`

3.5 - Lister les messages d’une conversation

Client request	
URL	`/messages/:userId`
Method	`GET`
Authorization	`Bearer security_token`
Server response	
Content-Type	`application/json`

Body	`[`
	`  {`
	`  id : number,`
	`  from_user : number,`
	`  to_user : number,`
	`  message : string`
	`  },`
	`  …`
	`]`

Response Code	`200 – OK`
	`401 – Unauthorized : invalid security token`
	`500 – Internal server error`

ANNEXE 4 – JSON WEB TOKEN

Qu’est-ce qu’un JWT ?

La réponse ici : https://jwt.io/introduction

Comment générer un JWT ?

Tout d'abord, vous aurez besoin d'installer le paquet `jsonwebtoken`. Techniquement, il est possible de tout gérer sans dépendance (un exercice que je vous recommande, simple et très instructif), mais ici, ça va finir pas être vraiment trop long.

Vous aurez également besoin des types du paquet pour une bonne prise en charge avec TypeScript :

```
npm i --save-dev @types/jsonwebtoken
```

Importer le paquet JsonWebToken

```
import JWT from "jsonwebtoken";
```

Créer un JWT

```
const token = JWT.sign(payloadData, privateKey);
```

Vérifier un JWT

```
const payloadData = JWT.verify(token, privateKey);
```

ANNEXE 5 – HACHAGE

L'API Web Crypto fournit tout un ensemble d'outils cryptographiques de bas niveau ne nécessitant pas de dépendance externe. Il est possible de manipuler de nombreux algorithmes, des AES aux RSA en passant par les courbes elliptiques.

ArrayBuffer

Comme indiqué plus tôt, Crypto fournit des outils bas niveau qui ne manipulent que des tableaux d'octets. Donc il n'est pas possible de manipuler directement des chaines de caractères. Il va donc falloir convertir les chaines de caractères en tableau d'octets (`ArrayBuffer`).

```
const textEncoder = new TextEncoder();
const arrayBuffer = textEncoder.encode(stringData);
```

Hachage

Il est ensuite possible d'appliquer un algorithme de hachage à ce tableau d'octets :

```
const hash = crypto.subtle.digest({name : "SHA-256"}, arrayBuffer);
```

Mais attention, le hash produit est lui aussi un tableau d'octets. Pour le rendre plus facilement manipulable, il faut le transformer en une chaine de caractères représentant sa valeur en base hexadécimale.

Conversion en hexadécimal

Le principe est simple, on prend chaque octet du tableau et on le transcrit en base 16. Si la valeur de l'octet ne tient que sur un chiffre, on ajoute un zéro devant :

```
const hexHash = Array.from(new Uint8Array(hash)).map((byte) => {
    return byte.toString(16).padStart(2, "0");
}).join("");
```