EPHEC
HAUTE ÉCOLE | TECH

T2112
Développement informatique III (Pratique)

Architecture REST, Middlewares, Documentation API & Frontend
TP 03

Auteur(s) :
Jonathan Noël

Dernière mise à jour le
16 février 2026

Table des matières

Légendes

👨‍🏫 Conseil du professeur.
ℹ️ Une information ...
🚨 Pré-requis pour le TP.
🖥️ Exercice à réaliser.
👓 Exercice d'observation ou de lecture.

1 Introduction

🚨 Pour réaliser ce TP, vous devez absolument avoir terminé le TP précédent. Si ce n'est pas le cas, c'est que vous êtes en retard et que vous devriez le rattraper au plus vite. Attention que tous les TP suivants auront comme prérequis le précédent.

Dans le TP précédent, vous avez mis en place la persistance des données avec une base de données et un ORM. Cependant, le fichier server.ts ou les fichiers de routes commencent probablement à contenir beaucoup de logique métier (requêtes DB, validation, réponse HTTP), ce qui rend le code difficile à maintenir.

👨‍🏫 MAINTENABILITÉ, MAINTENABILITÉ, MAINTENABILITÉ, MAINTENABILITÉ 😄

L'objectif de ce TP est de professionnaliser l'API en suivant les standards RESTful vus au cours théorique. Nous allons séparer les responsabilités, sécuriser les échanges via des middlewares et documenter l'API pour qu'elle soit consommable par le frontend.

Aussi, au lieu de continuer avec des pages HTML statiques limitées, nous allons faire un saut technologique aujourd'hui. Nous allons restructurer tout le projet pour séparer clairement le Backend (votre API) du Frontend (votre interface). Pour l'interface, nous utiliserons React, le framework le plus utilisé sur le marché pour créer des interfaces dynamiques.

Pour certaines séances de travail pratique, il est interdit d'utiliser les outils d'intelligence artificielle génératives afin que vous appreniez les fondamentaux correctement, ce qui est le cas de cette seconde séance. Aidez-vous un maximum des documentations officielles liées aux technologies que vous utiliserez dans le tp. Si dans les 30 dernières minutes du TP vous n'arrivez pas à avancer, aidez-vous de l'IAg, mais pas avant !

2 Partie A : Professionnalisation du Backend

2.1 Retrait du rouge qui se trouve partout dans le projet !

Pour retirer les erreurs concernant les imports, il faut mettre à jour TypeScript, même si le code fonctionne parfaitement bien. Mais un fichier rouge signifie généralement une réelle erreur, dans ce cas-ci ça peut être perturbant. Pour ce faire, mettez à jour le fichier `"typescript.json"` avec :

```
// Retirez
"module": "nodenext",
// Ajoutez
"moduleResolution": "node",
"allowSyntheticDefaultImports": true,
```

2.2 Architecture MVC : Les contrôleurs

Pour respecter le principe de séparation des responsabilités, la logique de traitement ne doit pas se trouver dans la définition des routes. Pour ce faire,

1. Créez un dossier `"src/controllers"`.
2. Extrayez la logique de vos routes existantes vers des fonctions exportées. Exemple :

```
// Avant extraction de logique : src/routes/userRoutes.ts
router.get("/", async (req: Request, res: Response) => {
  try {
    const users = await User.findAll();
    res.json(users);
  } catch (error) {
    res.status(500).json({ error: "Erreur lors de la récupération" });
  }
});
```

```ts
// Après extraction de logique : src/controllers/userController.ts
import type { Request, Response } from "express";
import User from "../models/User";

export const getAllUsers = async (req: Request, res: Response) => {
  try {
    const users = await User.findAll();
    res.status(200).json(users);
  } catch (error) {
    res.status(500).json({ error: (error as any).message });
  }
};
```

```ts
// Le routeur ne fait plus que lier une URL à une méthode du contrôleur
import * as userController from "../controllers/userController";

router.get("/", userController.getAllUsers);
```

ℹ️ Pensez à commit votre travail !

2.3 Les Middlewares

Pour rappel, un middleware est une fonction qui s'exécute avant que la requête n'arrive au contrôleur final (voir slides P3.1).

2.3.1 Création d'un Logger

Créez un dossier `"src/middlewares"` et un fichier `"logger.ts"`. Ce middleware affichera dans la console chaque requête reçue.

```ts
import { Request, Response, NextFunction } from 'express';

export const requestLogger = (req: Request, res: Response, next: NextFunction) => {
  console.log(`${new Date().toISOString()} - ${req.method} ${req.url}`);
  next(); // Indispensable pour passer à la suite !
};
```

👨‍🏫 Comprenez bien la fonction `next()` en lisant la documentation officielle et renseignez-vous sur d'autres méthodes utilisables à cet endroit.

Ensuite, dans `"server.ts"`, il faut activer le middleware globalement avec `app.use(requestLogger);` avant la déclaration des routes.

```
Base de données synchronisée
Serveur lancé sur http://localhost:3000
2026-02-16T10:56:56.627Z - GET /api/users
```

FIGURE 1 – Résultat du middleware logger

ℹ️ Pensez à commit votre travail !

2.4 Gestion centralisée des erreurs

Plutôt que de répéter des `try...catch` avec des `res.status(500)` partout, Express permet de gérer les erreurs à un seul endroit.

Créez un middleware de gestion d'erreurs - qui prend 4 paramètres (err, req, res, next) - dans `"src/middlewares/errorHandler.ts"`. Ce middleware affiche l'erreur en console via `console.error(...)`, récupère le statut de l'erreur et le message de l'erreur pour ensuite définir comme résultat de la

requête le status dynamiquement récupéré et en JSON le message reçu. Appelez ensuite ce middleware après toutes les routes dans server.ts, ça doit être la dernière étape !

👨‍🏫 `res.status(...).json({...})`

ℹ️ Pensez à commit votre travail !

2.5 Documentation avec Swagger

Une API doit être documentée pour plusieurs raisons : listing des routes disponibles, des données requises (token d'authentification, paramètres, etc.) ou simplement pour un partage d'informations avec les collègues. Imaginez bien que dans le monde du développement d'applications, le frontend et le backend n'évoluent pas forcément en parallèle en fonction des équipes frontend et backend. Afin de documenter l'API, nous utiliserons la spécification OpenAPI (via Swagger UI).

2.5.1 Installation

```
npm install swagger-ui-express swagger-jsdoc
npm install @types/swagger-ui-express @types/swagger-jsdoc --save-dev
```

2.5.2 Configuration

Ajoutez la configuration dans un fichier dédié `"src/config/swagger.ts"` :

```
// Dans src/config/swagger.ts
import swaggerJsdoc from "swagger-jsdoc";

const swaggerOptions = {
  definition: {
    openapi: "3.0.0",
    info: {
      title: "Mon API Géniale",
      version: "1.0.0",
    },
  },
  // Chemin vers les fichiers contenant les annotations
  apis: ["./src/routes/*.ts"],
};

export const swaggerSpec = swaggerJsdoc(swaggerOptions);

// Dans les imports de server.ts
import swaggerUi from "swagger-ui-express";
import { swaggerSpec } from "./config/swagger";
// En premier de la liste des routes server.ts
app.use("/api-docs", swaggerUi.serve, swaggerUi.setup(swaggerSpec));
```

Si vous lancez le serveur et que vous accédez à `"http://localhost:3000/api-docs/"`, vous devriez avoir l'interface suivante (figure 2).

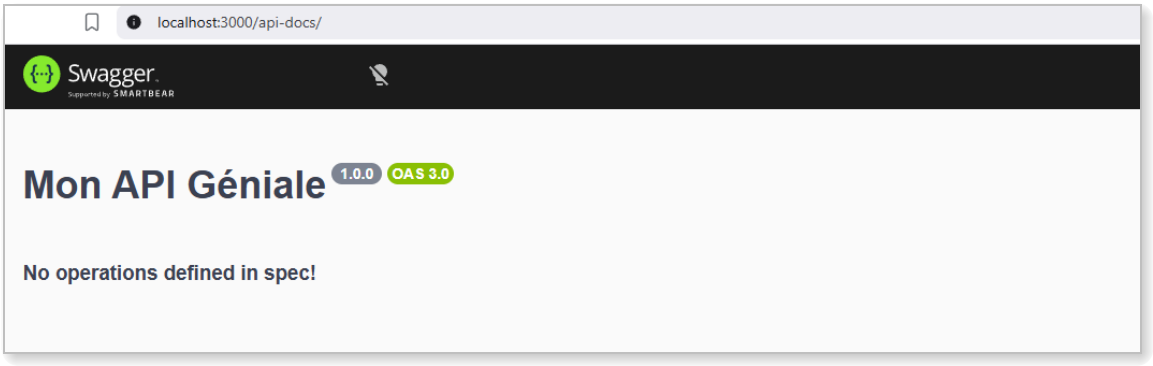

FIGURE 2 – Swagger fonctionne, youpi !

2.5.3 Annotation

Utilisez JSDoc au-dessus de vos routes pour générer la doc :

👨‍🏫 Attention que l'indentation est très importante, sous peine que la doc ne se génères pas !

```
/**
 * @swagger
 * /api/users:
 *  get:
 *    summary: Récupère la liste des utilisateurs
 *    tags: [Users]
 *    responses:
 *      200:
 *        description: Succès
 */
router.get('/', userController.getAllUsers);
```

FIGURE 3 – Première documentation définie !

ℹ️ Pensez à commit votre travail !

2.6 Préparation à la consommation React (CORS)

Par défaut, un navigateur interdit à un site A (votre futur Frontend) de récupérer des données d'un site B (votre Backend) s'ils ne sont pas sur le même port. C'est la sécurité CORS (Cross-Origin Resource Sharing).

Pour autoriser votre futur Frontend à discuter avec le Backend, il faut installer la librairie permettant de définir les règles cors et ajouter ces règles dans le fichier server.ts.

```
npm install cors (et @types/cors en dev).
```

```
// A définir dans server.ts
import cors from 'cors';
app.use(cors()); // Autorise tout le monde (acceptable uniquement en dev)
```

ℹ️ Pensez à commit votre travail !

2.7 Exercices

2.7.1 Refactoring complet

Si ce n'est pas encore fait, restructurez l'intégralité de votre projet pour utiliser des contrôleurs. Aucune logique métier ne doit résider dans `"src/routes"`. Vos routes doivent être "propres".

ℹ️ Pensez à commit votre travail !

2.7.2 Middleware de validation

Créez un middleware nommé `checkIdParam`. Il doit s'assurer que lorsqu'une route attend un paramètre `:id` (ex : `"/users/12"`), cet ID est bien un nombre entier valide. Si l'ID n'est pas valide, le middleware doit renvoyer une erreur 400 Bad Request immédiatement, sans appeler le contrôleur.

Appliquez ce middleware sur vos routes `GET /:id`, `PUT /:id` et `DELETE /:id`.

ℹ️ Pensez à commit votre travail !

2.7.3 Documentation complète

Documentez toutes les routes de votre API (`GET, POST, PUT, DELETE`) avec Swagger :

— Précisez les paramètres attendus (path parameters, body).

— Définissez les types de réponses possibles (200, 201, 400, 404, 500).

— Vérifiez le résultat sur `"http://localhost:3000/api-docs"`.

ℹ️ Pensez à commit votre travail !

3 Partie B : Architecture Full Stack

C'est le grand moment de ménage. Nous allons créer une structure de type "Monorepo". C'est un répertoire qui contient aussi bien le code source frontend et backend.

3.1 Restructuration des dossiers

1. À la racine du projet, créez un dossier nommé `server`.
2. Déplacez **TOUT** ce qui concerne votre backend actuel (`src`, `package.json`, `tsconfig.json`, `.env`, `dist`) à l'intérieur de ce dossier `server`.

   👨‍💻 Soyez smart en ne déplacant pas le dossier `"node_modules"` ! C'est un dossier très lourd, analysez sa taille. Refaite un `npm install` dans le dossier server pour réinstaller les dépendances.

3. Exception : Gardez le dossier `.git` à la racine.

La structure doit ressembler à ceci :

```
dev3-backend-tp1/
|-- server/      (Votre API Node.js)
|-- .git/        (Votre gestion de version)
```

👨‍💻 Attention : Pour lancer votre backend, vous devrez désormais faire `cd server` dans votre terminal avant de faire `npm start`.

ℹ️ Pensez à commit votre travail ! Votre git est desormais utiliasble à la racine du projet et non dans le dossier server.

3.2 Initialisation du client (Vite + React)

Nous allons générer le squelette du Frontend avec l'outil **Vite**. Il permet de générer un squelette complet avec une base de code et une architecture React saine.

1. Placez-vous à la racine du projet (`dev3-backend-tp1/`).
2. Exécutez la commanda afin de générer le projet frontend :

```
npm create vite@latest client -- --template react-ts
```

3. Cela crée un dossier `client`. Installez ses dépendances et lancez-le :

```
cd client
npm install
npm run dev
```

Votre Frontend tourne désormais sur http://localhost:5173. Vous avez donc deux serveurs lancés en parallèle (ceci nécessite deux terminaux ouverts).

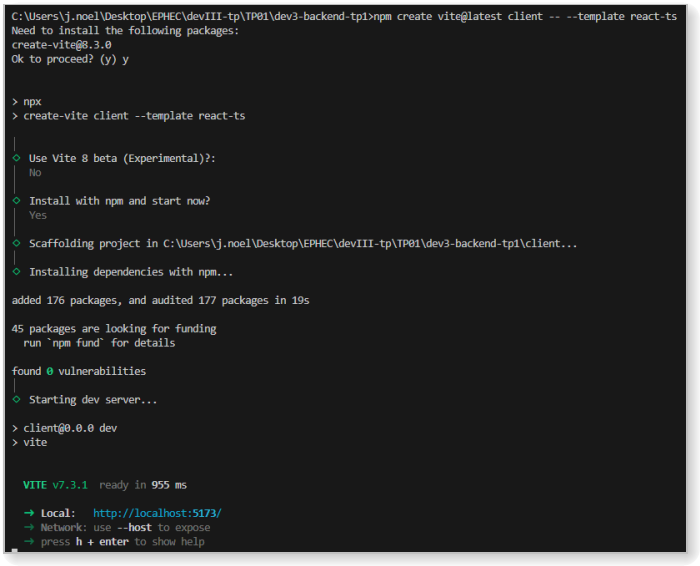

FIGURE 4 – Processus de génération de projet React via Vite (Zommez sur l'image)

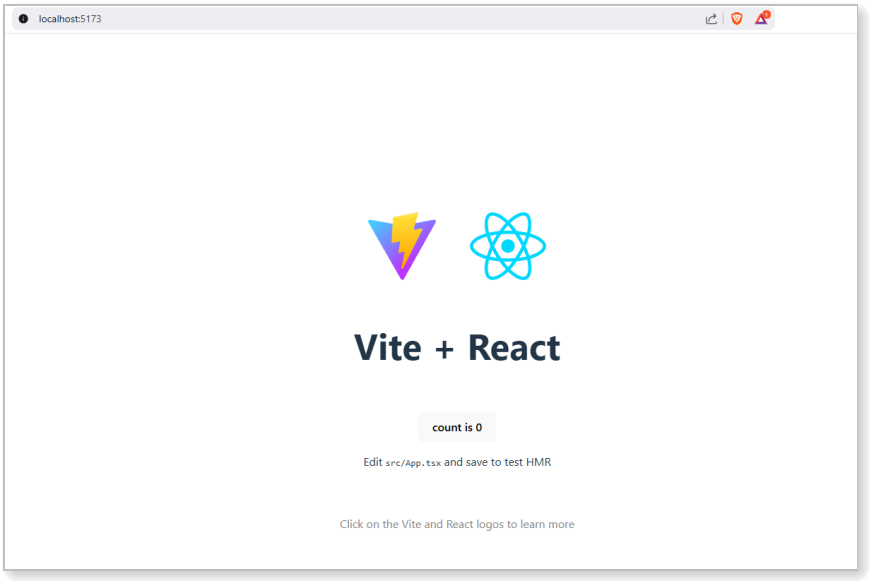

FIGURE 5 – Page de base React sur http://localhost:5173

ℹ️ Pensez à commit votre travail !

3.3 Pour se simplifier la vie - Concurrently

Gérer deux terminaux séparés (un pour le serveur, un pour le client) peut devenir fastidieux au quotidien. Nous allons automatiser le démarrage des deux environnements avec une seule commande grâce à la librairie `concurrently`.

A la racine de votre projet, c.à.d. à l'endroit ou se trouvent les dossiers frontend et backend, on va initialiser un `"package.json"` afin d'installer une librairie qui s'appelle `concurrently` et qui permet de lancer les 2 services en concurrence.

1. Placez-vous à la **racine** de votre projet global.
2. Si ce n'est pas déjà fait, initialisez un fichier `package.json` pour la racine :

   ```
   npm init -y
   ```

3. Installez le paquet `concurrently` en dépendance de développement :

   ```
   npm install concurrently --save-dev
   ```

4. Ouvrez le `package.json` **de la racine** et modifiez la section `"scripts"`. Nous utilisons l'option `--prefix` pour dire à npm d'exécuter les commandes dans les sous-dossiers correspondants.

```
{
  "name": "mon-super-beau-projet",
  "scripts": {
    "//": "Installe les dépendances des deux projets",
    "setup": "npm install --prefix server && npm install --prefix client",

    "//": "Raccourcis vers les commandes des sous-dossiers",
    "start:server": "npm run dev --prefix server",
    "start:client": "npm run dev --prefix client",

    "//": "Lance les deux en parallèle",
    "dev": "concurrently \"npm run start:server\" \"npm run start:client\""
  },
  "devDependencies": {
    "concurrently": "x.y.z" // Dernière version, celle qui s'installe par défaut
  }
}
```

Désormais, vous n'avez plus besoin que d'un seul terminal ouvert à la racine du projet pour tout lancer.

```
npm run dev
```

3.3.1 Rechargement du "server" à chaque modification

Pour le moment, vous devez relancer le serveur express après chaque modificiation. Cela n'est pas comfortable lorsqu'on travaille dessus. Pour ce faire, il existe une option sur tsx permettant ce rechargement, appelé "Watch Mode".

```
// Adaptez votre script "start" dans le package.json de "server"
"start": "tsx watch src/server.ts"
```

ℹ️ Pensez à commit votre travail !

4 Partie C : Connexion Client-Serveur

React fonctionne avec des **Composants** (fichiers `.tsx`). Le point d'entrée est généralement `src/App.tsx`.

Pour afficher vos données Backend dans React, nous utiliserons deux "Hooks" fondamentaux :

— **`useState`** : Pour stocker les données (mémoire du composant).

— **`useEffect`** : Pour lancer l'appel réseau au chargement du composant.

Concernant l'apprentissage de React, référez-vous à la cheat sheet [1]. Elle couvre un ensemble de concepts fondamentaux pour démarrer avec le langage.

Exemple de pattern à reproduire dans App.tsx :

```tsx
import { useEffect, useState } from "react";

// Définition d'une interface pour le typage
// Sera couvert plus en profondeur en TH
interface User {
    id: number;
    name: string;
}

function App() {
  // 1. Définition de l'état
  const [data, setData] = useState<User[]>([]);

  // 2. Appel API au montage du composant
  useEffect(() => {
    fetch("http://localhost:3000/api/users")
      .then(res => res.json())
      .then(result => setData(result))
      .catch(err => console.error(err));
  }, []);

  // 3. Rendu (JSX)
  return (
    <div>
      <h1>Liste des utilisateurs</h1>
      <ul>
        {data.map((item) => (
          <li key={item.id}>{item.name}</li>
        ))}
      </ul>
    </div>
  );
}

export default App;
```

ℹ️ Pensez à commit votre travail !

ℹ️ A ce moment, il est intéressant de push votre travail vers le serveur distant.

1. https://www.geeksforgeeks.org/reactjs/react-cheat-sheet/

5 Exercices

5.1 Backend Clean-up

— Assurez-vous que votre Backend est entièrement migré dans le sous-dossier `/server`.

— Refactorez vos routes pour utiliser des **Contrôleurs**.

— Vérifiez que **Swagger** est accessible sur `"lhttp://localhost:3000/api-docs"` et documentez l'ensemble de vos routes, vous n'en avez pas encore beaucoup. Ensuite, pour chaque nouvelle route, il est indispensable des les documenter.

ℹ️ Pensez à commit votre travail !

5.2 Frontend Setup

— Dans le dossier `/client`, l'objectif sera de recréer votre interface finale du TP02 en utilisant React.

🧑‍💻 Créez une interface TypeScript pour représenter votre modèle de données (User) et ensuite récupérer la liste des users via `fetch()`. Après cela, générez vos `<ul><li>` sur base des données reçues.

— Il faudra évidemment nettoyer le fichier `App.tsx` et `App.css` pour retirer le code de démonstration de Vite (compteur, logos, etc.).

ℹ️ Pensez à commit votre travail !

ℹ️ A ce moment, il est intéressant de push votre travail vers le serveur distant.