Rapport de développement
« API rest express.js »

Projet réalisé et développer
par Théo Michellon

Sommaire

Table des matières

I) 🚀 Contexte

Ce projet consiste à développer une API en ExpressJS qui gère deux types de données : les Pokémon, retournant des données au format JSON. Ce projet peut être réalisé individuellement ou en groupe.

II) 🎯 Objectifs

API Pokémon (JSON): Créer des routes pour gérer les informations sur les Pokémon (CRUD) et retourner les réponses en format JSON. 🐾
Pour les attributs des Pokémon, vous pouvez vous référer aux exemples ci-dessous :

 # Exemple de données pour un Pokémon

```
{
  "id": 1,
  "name": "Bublizarre",
  "type": "Grass",
  "level": 5,
  "evolution": "Herbizarre"
}
```

III) ✓ Exigences

Utiliser ExpressJS pour le backend. 💎
L'API doit respecter les principes RESTful. 🔄
Documenter clairement les routes, incluant des exemples de requêtes et de réponses pour chaque type de données. 📄

IV) Réalisation du projet

Le projet a été réalisé selon les étapes suivantes :

a) Installation du projet
b) Installation des modules
c) Configurateur du serveur Express
d) Configuration de la base de données
e) Ajout des routes
f) Gestion des erreurs
g) Test API

a) Installation du projet

Création d'un nouveau répertoire pour le projet. Depuis une invite de commande à l'intérieur. Exécutez la commande suivante pour initialiser votre projet avec npm (le gestionnaire de paquets Node.js).

```
npm init -y
```

b) Installation des modules

Installation du framework Express.js en tant que dépendance du projet en exécutant la commande suivante :

```
npm install express
```

c) Configurateur du serveur Express

Création d'un fichier app.js ou index.js pour configurer le serveur Express.

Pour créer et utiliser le serveur il faut saisir le code suivant :

```
const express = require('express');
const app = express();
const port = 3000;

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

app.listen(port, () => {
    consol.log(`Server listening at
http://localhost:${port}`);
});
```

d) Ajout des routes- CRUD

- Create (Créer) - Route POST :
  - Endpoint : /pokemons
  - Description : Cette route est utilisée pour créer une nouvelle ressource. Les données sont envoyés sous le format JSON
- Read (Lire) - Routes GET :
  - Endpoint : /pokemons (ou /pokemons/:id pour une ressource spécifique)
  - Description : Ces routes permettent de récupérer des informations sur une ou plusieurs ressources. Si aucun identifiant n'est spécifié alors toutes les données de la route sont renvoyées
- Update (Mettre à jour) - Route PUT ou PATCH :
  - Endpoint : /pokemons/:id
  - Description : Cette route est utilisée pour mettre à jour une ressource existante.
- Delete (Supprimer) - Route DELETE :
  - Endpoint : /pokemons/:id
  - Description : Cette route est utilisée pour supprimer une ressource spécifique.

Au cours du projet j'ai donc réalisé les routes suivantes :

- ❖ POST /pokemons : Créer un nouveau pokemon.
- ❖ GET /pokemons : Récupérer la liste complète des pokemons.
- ❖ GET /pokemons /:id : Récupérer un pokemon en particulier.
- ❖ PUT /pokemons /:id : Modifier les données d'un pokemon existante.
- ❖ DELETE /pokemons /:id : Supprimer un pokemon spécifique.

e) Configuration de la base de données

Sequelize est un ORM TypeScript et Node.js moderne pour Oracle, Postgres, MySQL, MariaDB, SQLite et SQL Server, et plus encore. Doté d'une prise en charge solide des transactions, de relations, d'un chargement rapide et paresseux, d'une réplication en lecture et bien plus encore.

Installation de Sequelize :

```
npm install sequelize sqlite3
```

Configuration de Modèles

```
import { Sequelize, DataTypes } from 'sequelize';

// Configuration de la base de données SQLite3
const sequelize = new Sequelize({
  dialect: "sqlite",
  storage: "database.sqlite", // Nom du fichier de base de données
});

// Définition du modèle Pokemon
const Pokemon = sequelize.define("Pokemon", {
  name: {
    type: DataTypes.STRING,
    allowNull: false,
  },
  pokemon_type: {
    type: DataTypes.STRING,
    allowNull: false,
  },
  size: {
    type: DataTypes.FLOAT,
    allowNull: false,
  },
  weight: {
    type: DataTypes.FLOAT,
    allowNull: false,
  },
  date_capture: {
    type: DataTypes.DATE,
    allowNull: false,
  },: DataTypes.DATE,
});
```

Exemple d'utilisation :

```
// Route pour créer un nouveau Pokémon (POST request)
app.post("/pokemons", async (req, res) => {
    const newPokemon = await Pokemon.create(req.body);
    res.status(201).json(newPokemon);
});

// Route pour obtenir les détails d'un Pokémon par ID (GET request)
app.get("/pokemons/:id", async (req, res) => {
    const pokemonId = parseInt(req.params.id);
    const pokemon = await Pokemon.findByPk(pokemonId);

    if (pokemon) {
      res.json(pokemon);
    } else {
      res.status(404).send("Pokemon not found");
    }
});
```

f) Gestion des erreurs

Il existe de nombreuses manière de gérer les erreurs dans le code pour ma part j'ai développé la gestion d'erreur la plus simple possible.

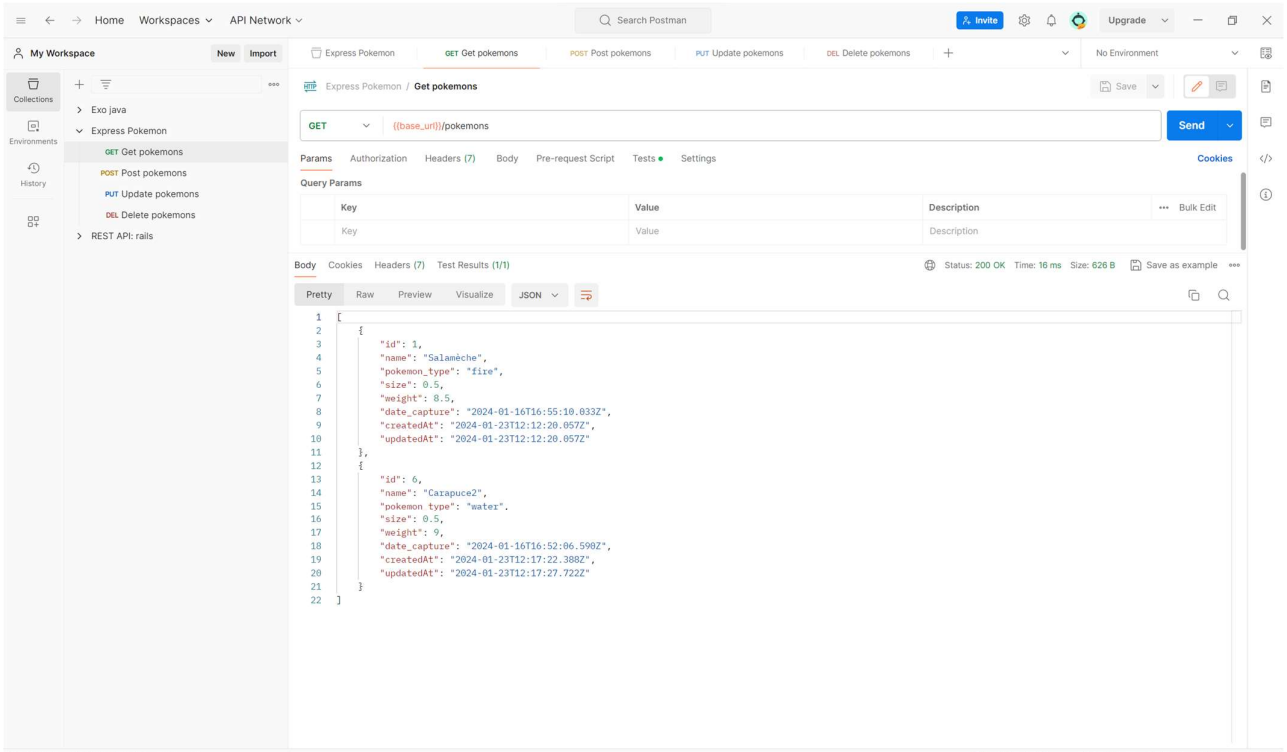
- Utilisation de try/catch
- Renvoi de réponse avec erreur lors de valeur non trouvé par exemple

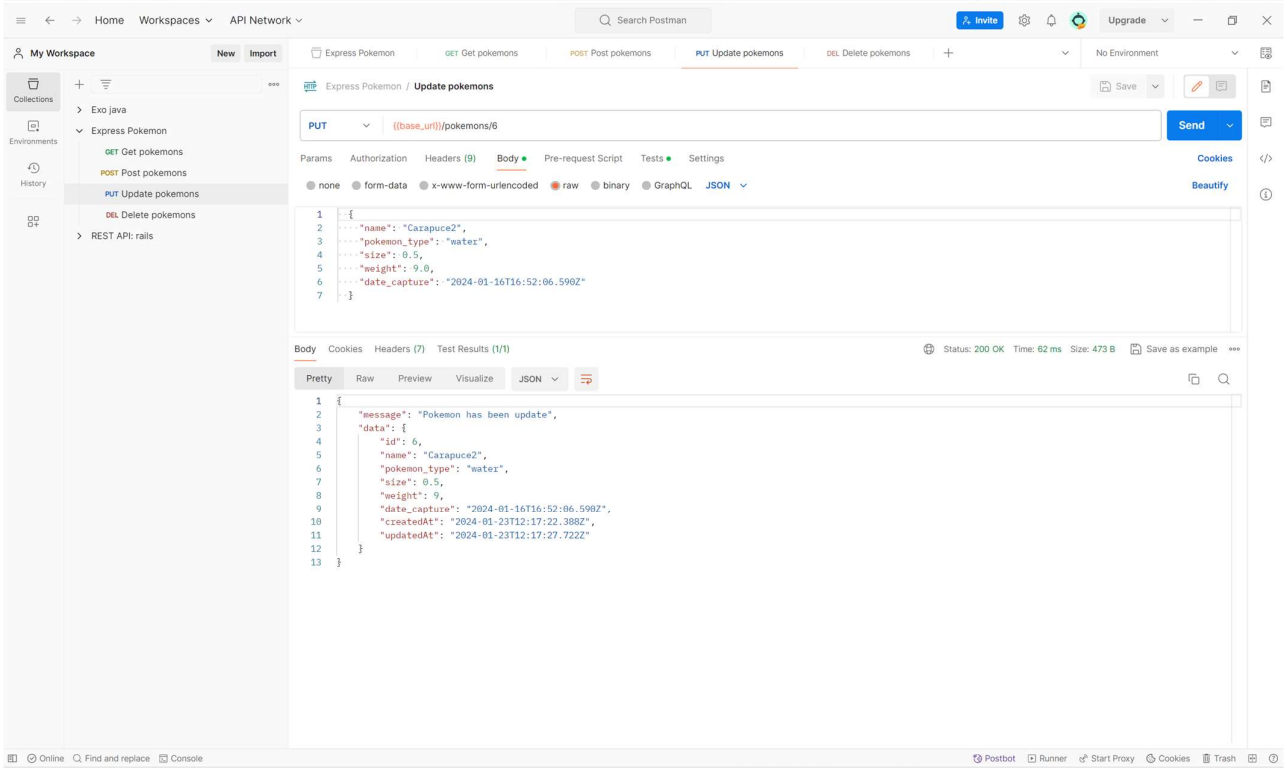
```
// Route pour mettre à jour un Pokémon par ID (PUT request)
app.put("/pokemons/:id", async (req, res) => {
  try {
    const pokemonId = parseInt(req.params.id);
    const pokemon = await Pokemon.findByPk(pokemonId);

    if (pokemon) {
      // Mise à jour des attributs du Pokémon
      await pokemon.update(req.body);

      res.json({ message: "Pokemon has been update", data: pokemon });
    } else {
      res.status(404).send("Pokemon not found");
    }
  } catch (error) {
    console.error(error);
    res.status(500).send("Internal Server Error");
  }
});

// Route pour obtenir les détails d'un Pokémon par ID (GET request)
app.get("/pokemons/:id", async (req, res) => {
  try {
    const pokemonId = parseInt(req.params.id);
    const pokemon = await Pokemon.findByPk(pokemonId);

    if (pokemon) {
      res.json(pokemon);
    } else {
      res.status(404).send("Pokemon not found");
    }
  } catch (error) {
    console.error(error);
    res.status(500).send("Internal Server Error");
  }
});
```

g) Test API

Postman est une plateforme API permettant de créer et d'utiliser des API. Postman simplifie chaque étape du cycle de vie des API et rationalise la collaboration afin que vous puissiez créer de meilleures API plus rapidement.

Vous trouverez ci-dessous deux exemples d'utilisation de cette API.





V) A votre tour de jouer

Pour cela rien de plus simple installer le repository, importer le fichier de config Postman dans votre logiciel et amusez vous