# T2112 - Devoir 1
# Rapport

Nom : GAULTHIER MEURIE

17 février 2026

## Table des matières



## Résumé

Dans ce devoir, j'ai étendu l'application de gestion d'utilisateurs vue en TP afin d'illustrer concrètement les concepts d'API REST avec Express, de persistance via Sequelize/SQLite et d'intégration frontend minimale. L'objectif est de mettre en place une gestion plus riche des utilisateurs (rôle et statut actif/inactif), avec typage TypeScript, validation des données et échanges client–serveur cohérents, pour démontrer la maîtrise des cinq points demandés dans l'énoncé (endpoints, base de données, typage/séparation, validation/erreurs, interaction client–API).

## 1 Objectifs (ce que je souhaite atteindre)

Sur base du projet de TP, je me suis fixé les objectifs suivants, en lien direct avec la checklist de l'énoncé :

- **API & routes Express** : créer et améliorer plusieurs endpoints REST autour des utilisateurs, notamment un filtre sur les utilisateurs actifs et une action pour changer leur statut.
- **Base de données et seeds** : enrichir le modèle `User` dans la base SQLite (ajout d'un rôle et d'un statut `isActive`) et mettre en place un « seed » initial pour pré-remplir la table.

— **Typage TypeScript et séparation** : structurer le code backend en couches claires (models / services / routes) avec un typage strict, afin de séparer la logique métier de la couche HTTP.
— **Validation des entrées et erreurs** : valider les données reçues (côté serveur et côté client) et retourner des réponses HTTP adaptées (400, 404, 500) avec des messages explicites.
— **Interaction client ↔ API** : faire communiquer la page HTML (formulaire + liste) avec l'API via `fetch`, en mettant à jour l'UI dynamiquement (création, suppression, activation/désactivation) et en affichant les erreurs de validation.

# 2 Stratégie et raisonnement

J'ai choisi de conserver la pile technologique des TP (Express, Sequelize, SQLite, TypeScript) pour rester cohérent avec le cours et me concentrer sur la qualité de l'architecture plutôt que sur de nouvelles dépendances. Côté backend, j'ai adopté une séparation claire en trois couches : le modèle Sequelize `User` pour la persistance, un service `userService` qui contient la logique métier (validation, création, suppression, changement de statut), et les routes Express qui exposent cette logique via des endpoints HTTP.

D'un point de vue base de données, j'ai décidé d'étendre la table `users` avec deux nouvelles colonnes : un rôle (`student` ou `teacher`) et un booléen `isActive`. La synchronisation avec SQLite se fait via `sequelize.sync({ alter: true })`, ce qui permet d'ajuster le schéma en fonction du modèle TypeScript. Pour disposer de données de test cohérentes, j'ai ajouté un petit seed au démarrage de l'application qui insère trois utilisateurs de démonstration si la table est vide.

Pour la validation des entrées, j'ai fait le choix de la gérer à la fois côté serveur (dans le service) et côté client (dans `script.js`). Côté serveur, une classe `HttpError` me permet de distinguer les erreurs fonctionnelles (400, 404) des erreurs internes (500) et de renvoyer un JSON d'erreur lisible. Côté client, je réutilise des règles similaires (nom/prénom obligatoires, regex pour les caractères autorisés) avant d'appeler l'API.

# 3 Réalisation (ce que j'ai réellement atteint et comment)

## 3.1 Fonctionnalités ajoutées

**API & routes Express.** J'ai enrichi l'API existante avec plusieurs endpoints autour des utilisateurs :

— **GET /api/users** : retourne la liste complète des utilisateurs en base (service `findAllUsers()`, annexe 4.2).
— **GET /api/users/active** : retourne uniquement les utilisateurs dont le champ `isActive` est à `true`.
— **POST /api/users** : crée un nouvel utilisateur après validation des champs `nom`, `prenom` et `role`. En cas d'erreur de validation, la route renvoie un code `400` et un message JSON explicite (annexe 4.1).
— **PATCH /api/users/ :id/toggle-active** : bascule le statut actif/inactif d'un utilisateur existant, renvoie `404` si l'ID n'existe pas.
— **DELETE /api/users/ :id** : supprime un utilisateur en base et renvoie un message de confirmation, ou un `404` si l'utilisateur n'est pas trouvé.

**Base de données, modèle et seed.** Le modèle Sequelize `User` (fichier `src/models/User.ts`, annexe 4.3) a été étendu avec les attributs suivants :

— `role` : chaîne de caractères représentant le rôle (`student` ou `teacher`), avec une valeur par défaut.

— `isActive` : booléen indiquant si l'utilisateur est actif ou non, initialisé à `true`.

La synchronisation avec SQLite se fait via `sequelize.sync({ alter: true })` dans `src/server.ts`. Une fonction `seedInitialUsers()` insère trois utilisateurs de démonstration si la table est vide, ce qui réalise un seed minimal pour faciliter les tests (annexe 4.4).

**Typage TypeScript et séparation des responsabilités.** Le code backend est structuré en trois couches :

— **Modèle** : `User` dans `src/models/User.ts`, qui définit la structure de la table et les types associés (notamment le type `UserRole`).

— **Service** : `src/services/userService.ts`, qui contient la logique métier (validation, création, suppression, filtrage par statut, etc.).

— **Routes** : `src/routes/userRoutes.ts`, qui se contente d'exposer les fonctions du service via des endpoints HTTP et de gérer les codes de retour.

**Validation et gestion des erreurs.** Dans `userService.ts`, une classe `HttpError` est utilisée pour représenter les erreurs fonctionnelles avec un code HTTP associé (400, 404). Les fonctions de création et de modification valident les champs :

— **Nom/prénom** : non vides, et ne contenant que des lettres (y compris accentuées), apostrophes et tirets, grâce à une expression régulière commune.

— **Rôle** : doit être soit `"student"`, soit `"teacher"`, sinon une erreur 400 est levée.

Les routes Express interceptent ces erreurs et renvoient un JSON d'erreur structuré avec le bon status. En cas d'exception non prévue, une erreur 500 générique est renvoyée.

**Interaction client–API et UI.** Côté frontend, le fichier `public/index.html` contient un formulaire avec trois champs (nom, prénom, rôle) et une carte listant les utilisateurs. Le fichier `public/script.js` gère :

— le chargement initial des utilisateurs via `fetch('/api/users')`, l'affichage dans une liste dynamique, et les boutons « Activer/Désactiver » et « X » pour la suppression ;

— la soumission du formulaire, la validation côté client, puis l'appel à `POST /api/users` si les données sont correctes ;

— l'affichage d'un message d'erreur sous le formulaire en cas de problème de validation côté client.

## 3.2 Exemple de code

L'exemple suivant illustre un des endpoints ajoutés, qui exploite la couche de service pour récupérer tous les utilisateurs et retourner une réponse HTTP correctement typée.

Listing 1 – Endpoint GET /api/users utilisant le service

```
router.get("/api/users", async (_req, res) => {
  try {
    const users = await findAllUsers();
    res.status(200).json(users);
  } catch (error) {
    console.error(error);
    res.status(500).json({ error: "Erreur lors de la r cup ration des
       utilisateurs." });
  }
});
```

### 3.3 Captures d'écran (description textuelle)

Au lieu d'inclure des captures d'écran, je décris ici les résultats observés lors des tests.

Après avoir lancé le serveur (`npm start`) et ouvert la page principale, l'interface se compose de deux blocs : un formulaire à gauche et une liste d'utilisateurs à droite. Le formulaire contient trois champs (nom, prénom, rôle) et un bouton « Ajouter l'utilisateur ». La liste affiche, pour chaque utilisateur, le nom et le prénom en gras, suivis d'une ligne plus petite indiquant le rôle (Étudiant ou Professeur), le statut (Actif ou Inactif) ainsi que l'ID technique. À droite de chaque ligne, deux boutons sont visibles : un bouton « Activer/Désactiver » et un bouton « X » pour la suppression.

Lorsque je crée un nouvel utilisateur avec des données valides, le formulaire se vide après soumission et la nouvelle entrée apparaît immédiatement dans la liste sans rechargement de la page. Le rôle et le statut « Actif » sont correctement affichés. Si je clique ensuite sur le bouton « Désactiver », le texte du statut passe à « Inactif » et le libellé du bouton devient « Activer », ce qui confirme le bon fonctionnement de l'endpoint `PATCH /api/users/:id/toggle-active`. Des appels `curl` vers `GET /api/users` et `GET /api/users/active` montrent bien les codes HTTP 200 et les payloads JSON attendus.

## 4 Annexes

### 4.1 Code source - routes utilisateur

Listing 2 – Extrait de src/routes/userRoutes.ts

```
router.get("/api/users", async (_req, res) => {
  try {
    const users = await findAllUsers();
    res.status(200).json(users);
  } catch (error) {
    res.status(500).json({ error: "Erreur lors de la r cup ration des
       utilisateurs." });
  }
});

router.post("/api/users", async (req, res) => {
  try {
    const user = await createUser(req.body);
    res.status(201).json(user);
  } catch (error) {
    if (error instanceof HttpError) {
      res.status(error.status).json({ error: error.message });
      return;
    }
    res.status(500).json({ error: "Erreur lors de la cr ation de l'
       utilisateur." });
  }
});

// ... autres routes : GET /api/users/active, PATCH /api/users/:id/
   toggle-active, DELETE /api/users/:id
```

### 4.2 Code source - service utilisateur

Listing 3 – Extrait de src/services/userService.ts

```
export class HttpError extends Error {
```

```
  status: number;

  constructor(status: number, message: string) {
    super(message);
    this.status = status;
  }
}

const NAME_REGEX = /^[A-Za-z  -    -    -  ' -]+$/;

function validateName(value: string, fieldLabel: string): string {
  const trimmed = value.trim();
  if (!trimmed) {
    throw new HttpError(400, `${fieldLabel} est obligatoire.`);
  }
  if (!NAME_REGEX.test(trimmed)) {
    throw new HttpError(
      400,
      `${fieldLabel} ne peut contenir que des lettres (avec accents), l'
         apostrophe (') et le tiret (-).`
    );
  }
  return trimmed;
}

export async function createUser(input: CreateUserInput) {
  const nom = validateName(input.nom, "Nom");
  const prenom = validateName(input.prenom, "Pr nom");
  const role = validateRole(input.role);

  return User.create({ nom, prenom, role, isActive: true });
}

// ... autres fonctions : findAllUsers, findActiveUsers, toggleActive,
   deleteUser
```

### 4.3 Code source - modèle Sequelize User

Listing 4 – Extrait de src/models/User.ts

```
export type UserRole = "student" | "teacher";

class User extends Model<InferAttributes<User>, InferCreationAttributes<
   User>> {
  declare id: CreationOptional<number>;
  declare nom: string;
  declare prenom: string;
  declare role: UserRole;
  declare isActive: boolean;
}

User.init(
  {
    nom: { type: DataTypes.STRING, allowNull: false },
    prenom: { type: DataTypes.STRING, allowNull: false },
    role: { type: DataTypes.STRING, allowNull: false, defaultValue: "
       student" },
```

```
    isActive: { type: DataTypes.BOOLEAN, allowNull: false, defaultValue:
        true },
  },
  { sequelize, modelName: "User", tableName: "users", timestamps: false
     }
);
```

### 4.4 Code source - serveur et configuration DB

Listing 5 – Extrait de src/server.ts

```
async function seedInitialUsers() {
  const count = await User.count();
  if (count > 0) return;

  await User.bulkCreate([
    { nom: "Dupont", prenom: "Jean", role: "student", isActive: true },
    { nom: "Martin", prenom: "Sophie", role: "student", isActive: false
       },
    { nom: "Durand", prenom: "Claire", role: "teacher", isActive: true
       },
  ]);
}

sequelize
  .sync({ alter: true })
  .then(async () => {
    await seedInitialUsers();
    app.listen(port, () => {
      console.log(`Serveur en  coute  sur le port ${port}`);
    });
  });
```

### 4.5 Code source - Frontend (HTML/JS)

Listing 6 – Extrait de public/index.html

```
<form class="space-y-4">
  <div class="grid gap-4 sm:grid-cols-2">
    <!-- champs nom et pr nom -->
  </div>

  <div class="space-y-1.5">
    <label for="role">R le </label>
    <select id="role">
      <option value="student"> tudiant </option>
      <option value="teacher">Professeur</option>
    </select>
  </div>

  <button type="submit">Ajouter l'utilisateur</button>
  <p id="form-error" class="hidden">Nom et pr nom sont obligatoires.</p
     >
</form>
```

Listing 7 – Extrait de public/script.js

```
function loadUsers() {
```

```
  fetch('/api/users')
    .then(res => res.json())
    .then(users => {
      // ... cr ation dynamique des <li> avec r le et statut actif/
          inactif
    });
}

document.querySelector('form').addEventListener('submit', (e) => {
  e.preventDefault();
  // ... validation c  t   client (champs non vides + regex) puis POST /
      api/users
});

loadUsers();
```