TP – API MVC Express avec POO

Objectifs

- Structurer une API Express en **MVC minimal** : `routes → controllers → models`.
- Pratiquer la **POO** en JavaScript (classes, encapsulation, méthodes).
- Réaliser un **CRUD** en mémoire (sans base), **directement dans le Model**.
- Tester avec **Postman** et respecter les **codes HTTP**.

Prérequis

Node/npm, VS Code, Postman. Connaissances de base : endpoints / verbes HTTP / JSON.

PARTIE 1 — GUIDÉE : Ressource `users`

Étape 0 — Initialiser le projet

```
mkdir tp-mvc-poo-lite && cd tp-mvc-poo-lite
npm init -y
npm install express
npm install -D nodemon
```

Dans `package.json` :

```
"scripts": {
  "start": "node src/server.js",
  "dev": "nodemon src/server.js"
}
```

Crée la structure :

```
src/
  server.js
  routes/
    user.routes.js
  controllers/
    user.controller.js
  models/
    user.model.js
```

Étape 1 — Point d'entrée Express

`src/server.js`

```
const express = require("express");
const app = express();
const PORT = 3000;

app.use(express.json());

// Mount routes
const userRoutes = require("./routes/user.routes");
app.use("/api/users", userRoutes);

// Health
app.get("/api/status", (req, res) => {
  res.json({ status: "ok", time: new Date().toISOString() });
});

// 404
app.use((req, res) => res.status(404).json({ error: "Route inconnue" }));

// Error handler
app.use((err, req, res, next) => {
  console.error("💥 Erreur serveur:", err.message);
  res.status(500).json({ error: "Erreur interne serveur" });
});

app.listen(PORT, () => console.log(`✅ API prête sur http://localhost:${PORT}`));
```

Étape 2 — Model POO avec stockage en mémoire

Idée clé : le **Model** porte la **forme des données** (classe `User`) **et** les **opérations CRUD** via des **méthodes statiques** sur un tableau privé en mémoire.

`src/models/user.model.js`

```
class User {
  #id; #name; #age;

  constructor({ id, name, age }) {
    this.#id = id;
    this.setName(name);
    this.setAge(age);
  }

  // ---------- Validation + Factory ----------
  static create({ id, name, age }) {
    if (typeof name !== "string" || !name.trim()) {
      throw new Error("Name must be a non-empty string");
```

```
    }
    if (typeof age !== "number" || age < 0) {
      throw new Error("Age must be a positive number");
    }
    return new User({ id, name: name.trim(), age });
  }

  // ---------- Encapsulation ----------
  get id()  { return this.#id; }
  get name(){ return this.#name; }
  get age() { return this.#age; }

  setName(name) {
    if (typeof name !== "string" || !name.trim()) throw new Error("Invalid name");
    this.#name = name.trim();
  }
  setAge(age) {
    if (typeof age !== "number" || age < 0) throw new Error("Invalid age");
    this.#age = age;
  }

  toJSON() { return { id: this.#id, name: this.#name, age: this.#age }; }

  // ---------- "Persistance" en mémoire ----------
  static #data = [
    User.create({ id: 1, name: "Alice", age: 25 }),
    User.create({ id: 2, name: "Bob", age: 30 }),
  ];

  static nextId() { return Date.now(); }

  static findAll() {
    return this.#data.map(u => u.toJSON());
  }

  static findById(id) {
    const u = this.#data.find(u => u.id === id);
    return u ? u.toJSON() : null;
  }

  static createOne({ name, age }) {
    const user = User.create({ id: this.nextId(), name, age });
    this.#data.push(user);
    return user.toJSON();
  }

  static updateOne(id, dto) {
    const idx = this.#data.findIndex(u => u.id === id);
    if (idx === -1) return null;

    // Mise à jour avec validation via setters
    if (dto.name !== undefined) this.#data[idx].setName(dto.name);
```

```
    if (dto.age  !== undefined) this.#data[idx].setAge(dto.age);

    return this.#data[idx].toJSON();
  }

  static deleteOne(id) {
    const before = this.#data.length;
    this.#data = this.#data.filter(u => u.id !== id);
    return this.#data.length !== before; // true si supprimé
  }
}

module.exports = User;
```

Étape 3 — Controller (appels directs au Model)

`src/controllers/user.controller.js`

```
const User = require("../models/user.model");

exports.listUsers = (req, res, next) => {
  try {
    const { q, minAge, maxAge, limit = 50, offset = 0 } = req.query;
    let data = User.findAll();

    // Petits filtres côté controller (démo simple)
    if (q)      data = data.filter(u =>
u.name.toLowerCase().includes(String(q).toLowerCase()));
    if (minAge) data = data.filter(u => u.age >= Number(minAge));
    if (maxAge) data = data.filter(u => u.age <= Number(maxAge));

    const start = Number(offset), end = start + Number(limit);
    return res.status(200).json({ total: data.length, data: data.slice(start, end) });
  } catch (e) { next(e); }
};

exports.getUser = (req, res, next) => {
  try {
    const user = User.findById(Number(req.params.id));
    if (!user) return res.status(404).json({ error: "Utilisateur non trouvé" });
    return res.status(200).json(user);
  } catch (e) { next(e); }
};

exports.createUser = (req, res, next) => {
  try {
    const { name, age } = req.body;
    if (name === undefined || age === undefined) {
      return res.status(400).json({ error: "name (string) et age (number) sont requis" });
    }
```

```
    const created = User.createOne({ name, age });
    return res.status(201).json(created);
  } catch (e) { next(e); }
};

exports.updateUser = (req, res, next) => {
  try {
    const updated = User.updateOne(Number(req.params.id), req.body);
    if (!updated) return res.status(404).json({ error: "Utilisateur non trouvé" });
    return res.status(200).json(updated);
  } catch (e) { next(e); }
};

exports.deleteUser = (req, res, next) => {
  try {
    const ok = User.deleteOne(Number(req.params.id));
    if (!ok) return res.status(404).json({ error: "Utilisateur non trouvé" });
    return res.status(204).send();
  } catch (e) { next(e); }
};
```

Étape 4 — Routes REST

`src/routes/user.routes.js`

```
const router = require("express").Router();
const ctrl = require("../controllers/user.controller");

router.get("/", ctrl.listUsers);
router.get("/:id", ctrl.getUser);
router.post("/", ctrl.createUser);
router.put("/:id", ctrl.updateUser);
router.delete("/:id", ctrl.deleteUser);

module.exports = router;
```

Tests Postman (exemples)

- **GET** `http://localhost:3000/api/users`
- **GET** `http://localhost:3000/api/users/1`
- **POST** `http://localhost:3000/api/users`

  ```
  { "name": "Charlie", "age": 22 }
  ```

- **PUT** `http://localhost:3000/api/users/1`

```
{ "name": "Alice L.", "age": 26 }
```

- **DELETE** `http://localhost:3000/api/users/1`

Filtres :
`GET /api/users?q=al&minAge=20&maxAge=40&limit=1&offset=0`

Codes attendus : 200, 201, 204, 400, 404, 500.

PARTIE 2 — EN AUTONOMIE : Ressource `products`

Mission

Reproduire la même architecture **(model + controller + routes)** pour `products`.

Spécifications

- **Product** : `id: number`, `name: string`, `price: number`.
- CRUD complet :
  - `GET /api/products` (filtres : `q`, `minPrice`, `maxPrice`, `limit`, `offset`)
  - `GET /api/products/:id`
  - `POST /api/products` (name, price requis)
  - `PUT /api/products/:id`
  - `DELETE /api/products/:id`

Tâches

1. **Model** `product.model.js`
   - Classe `Product` avec champs privés `#name`, `#price`.
   - Méthodes statiques : `findAll`, `findById`, `createOne`, `updateOne`, `deleteOne`.
   - Tableau statique privé `#data` pré-rempli (2–3 produits).
   - Validations dans `create` + setters (ex: `price >= 0`).
2. **Controller** `product.controller.js`
   - Même logique que `user.controller.js` (filtres à minima dans `list`).
3. **Routes** `product.routes.js`
   - Monter dans `server.js` avec :

```
const productRoutes = require("./routes/product.routes");
app.use("/api/products", productRoutes);
```

Jeux de tests suggérés

- **POST** `/api/products`

```
{ "name": "Clavier", "price": 49.9 }
```

- **GET** `/api/products?q=cla&minPrice=30&maxPrice=60&limit=1&offset=0`
- **PUT** `/api/products/:id`

```
{ "name": "Clavier Pro", "price": 69.9 }
```

- **DELETE** `/api/products/:id`

Bonus (optionnel)

- Ajouter un **PATCH** partiel (`name` ou `price`).
- Centraliser quelques **helpers HTTP** (200/201/204/404) si tu veux alléger le controller.
- Passer en **ES Modules** (`"type": "module"`) et utiliser `import/export`.

Récap

- Tu maîtrises maintenant une **API Express en MVC minimal** avec **POO** et **CRUD en mémoire**.
- Prochain jalon : brancher une **vraie base** (PostgreSQL/Mongo), puis **auth JWT**, **tests** et **Swagger/OpenAPI**.