Cours MERN - Semaine 2

Structuration de l'API : Routes et Contrôleurs

Abdelweheb GUEDDES & Mohamed Ben Jazia / Ecole Polytechnique Sousse

24 septembre 2025

Table des matières

1 Objectifs Pédagogiques Détaillés

Cette semaine, nous passons d'un simple serveur à une véritable architecture d'API. À la fin de cette session, vous serez capable de :

- **Expliquer** les limites d'un fichier serveur monolithique et l'importance de la Séparation des Préoccupations (SoC).
- **Définir** précisément le rôle d'un routeur et d'un contrôleur dans une architecture Express.
- **Utiliser** l'objet `express.Router()` pour créer des modules de routes encapsulés.
- **Créer** un fichier de contrôleurs contenant la logique métier sous forme de fonctions exportées.
- **Connecter** les routes aux fonctions de leurs contrôleurs respectifs.
- **Restructurer** (refactoriser) une application Express existante pour adopter cette nouvelle architecture.
- **Intégrer** les modules de routes dans l'application principale `server.js` en utilisant `app.use()`.
- **Valider** avec Postman que l'API reste fonctionnelle après la restructuration.

2 Partie 1 : Concepts Architecturaux et Techniques (1h15)

2.1 Le Problème : Le Fichier Monolithique

Lors de la première séance, nous avons créé un fichier 'server.js' fonctionnel. C'est parfait pour démarrer, mais imaginez une application réelle avec des dizaines, voire des centaines de routes (pour les articles, les utilisateurs, les commentaires, les catégories, etc.). Le fichier 'server.js' deviendrait rapidement :

- **Illisible :** Des milliers de lignes de code sans organisation claire.
- **Difficile à maintenir :** Modifier une route pour les utilisateurs risquerait de casser quelque chose pour les articles.
- **Non-collaboratif :** Deux développeurs ne pourraient pas travailler en même temps sur les routes des articles et des utilisateurs sans conflits constants.

Ce phénomène est souvent appelé "code spaghetti". Notre objectif est de transformer ce plat de spaghettis en une boîte de rangement bien organisée.

2.2 La Solution : Séparation des Préoccupations (SoC)

Le principe de SoC, que nous avons introduit la semaine dernière, nous dicte la marche à suivre. Nous allons diviser notre code en fonction de ses responsabilités.

Architecture Cible : Notre application va désormais avoir trois niveaux de responsabilité distincts :

1. **Serveur ('server.js')** : Le chef d'orchestre. Son seul rôle est de démarrer le serveur, de configurer les middlewares globaux (comme 'express.json()') et de **déléguer** les requêtes aux bons modules de routes.
2. **Routeur ('/routes/*.js')** : L'aiguilleur. Un fichier de routes ne s'occupe que de diriger le trafic. Il dit : "Si je reçois une requête `GET` sur l'URL `/`, j'appelle la

fonction `getAllArticles` du contrôleur."

3. **Contrôleur (‘/controllers/*.js‘)** : L'ouvrier spécialisé. Un fichier de contrôleurs contient la logique métier. C'est lui qui fait le vrai travail : interroger la base de données, traiter les données, construire la réponse et l'envoyer.

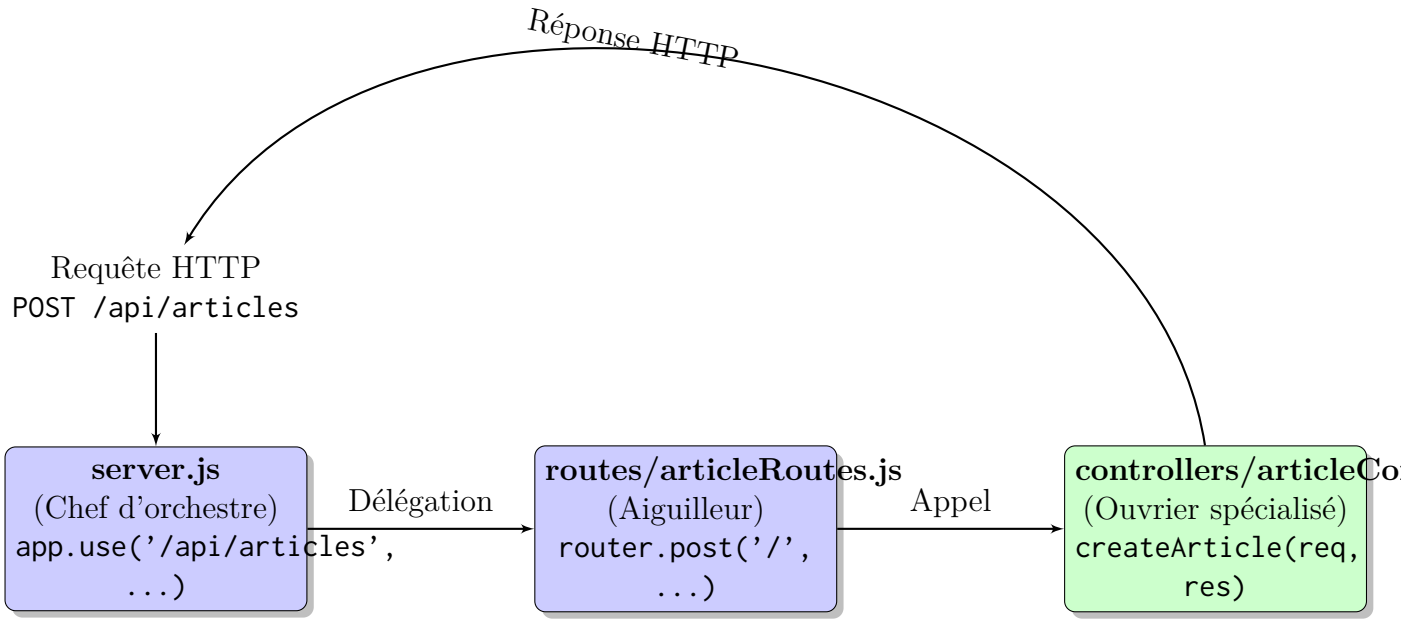

FIGURE 1 – Flux d'une requête dans une architecture structurée.

2.3 L'Outil Clé : `express.Router()`

Express nous fournit un outil puissant pour créer ces "aiguilleurs" : le routeur. Un `express.Router()` est une sorte de "mini-application" Express. On peut y attacher des routes et des middlewares, tout comme sur l'objet ‘app‘ principal. L'avantage est qu'on peut ensuite exporter ce routeur et l'importer dans notre application principale.

Cela nous permet d'encapsuler toute la logique de routage d'une ressource (comme les articles) dans un seul fichier, le rendant modulaire et réutilisable.

3 Partie 2 : Atelier Pratique - Refactorisation (1h45)

Nous allons prendre notre projet de la Semaine 1 et le restructurer.

3.1 Étape 1 : Préparation de l'Architecture

1. Reprenez votre projet 'mon-api-blog' de la semaine dernière.
2. À la racine du projet, créez les deux dossiers nécessaires à notre nouvelle architecture :

```
mkdir routes
mkdir controllers

```

Listing 1 – Création de la nouvelle arborescence

3.2 Étape 2 : Création du Contrôleur d'Articles

Commençons par isoler la logique métier.

1. Créez un nouveau fichier : `controllers/articleController.js`.
2. **Déplacez la logique.** Coupez les fonctions de rappel (callbacks) de vos routes `GET /api/test` et `POST /api/articles` depuis 'server.js' et collez-les dans ce nouveau fichier.
3. **Transformez-les en fonctions nommées et exportées.**

```
// Pour l'instant, nous n'avons pas de vraie logique, mais nous préparons la structure.

// Controleur pour la route de test (anciennement GET /api/test)
const testApi = (req, res) => {
    res.status(200).json({ message: 'Le test depuis le controleur a fonctionné !', success: true });
};

// controleur pour créer un article (anciennement POST /api/articles)
const createArticle = (req, res) => {
    const articleData = req.body;
    console.log('Données reçues par le controleur :', articleData);

    res.status(201).json({
        message: 'Article créé avec succès via le controleur !',
        article: { id: Date.now(), ...articleData }
    });
};

// On exporte les fonctions pour pouvoir les utiliser dans nos routes
module.exports = {
    testApi,
    createArticle
```

```
};

```

Listing 2 – controllers/articleController.js

3.3 Étape 3 : Création du Routeur d'Articles

Maintenant, créons l'aiguilleur qui utilisera ces fonctions.

1. Créez un nouveau fichier : `routes/articleRoutes.js`.
2. Dans ce fichier, nous allons :
   — Importer Express et créer une instance de `Router`.
   — Importer nos nouvelles fonctions depuis le contrôleur.
   — Définir nos routes sur le routeur, en les associant aux fonctions du contrôleur.
   — Exporter le routeur.

```
const express = require('express');
const router = express.Router();

// On importe les fonctions du controleur
const { testApi, createArticle } = require('../controllers/
    articleController');

// Définition des routes
// Note : Le chemin '/' ici correspondra à la racine de ce que
    nous définirons dans server.js

// Route GET pour /api/test (devient /test dans ce routeur)
router.get('/test', testApi);

// Route POST pour /api/articles (devient / dans ce routeur)
router.post('/', createArticle);

// On exporte le routeur pour l'utiliser dans server.js
module.exports = router;

```

Listing 3 – routes/articleRoutes.js

3.4 Étape 4 : Mettre à jour le Serveur Principal

Il est temps de nettoyer 'server.js' et de le transformer en chef d'orchestre.

1. Ouvrez 'server.js'.
2. **Supprimez les anciennes définitions de routes** ('app.get('/api/test', ...)' et 'app.post('/api/articles', ...)').
3. **Importez votre nouveau routeur.**
4. **Utilisez `app.use()`** pour dire à Express d'utiliser ce routeur pour toutes les requêtes qui commencent par un certain chemin (par ex., `/api/articles`).

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

```

```
// On importe notre nouveau routeur
const articleRoutes = require('./routes/articleRoutes');

// Middleware pour parser le JSON
app.use(express.json());

// Route GET de base (reste ici car elle est générale)
app.get('/', (req, res) => {
  res.status(200).send('<h1>Page d_accueil de notre API de Blog !</h1>');
});

// --- NOUVEAU : Utilisation du routeur ---
// Express utilisera le routeur 'articleRoutes' pour toute requête
// commençant par '/api/articles'
app.use('/api/articles', articleRoutes);

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

```

Listing 4 – server.js - Version refactorisée

Point Clé : Dans 'server.js', on a 'app.use('/api/articles', articleRoutes)'. Dans 'articleRoutes.js', on a 'router.post('/', createArticle)'. Express combine les deux. Une requête `POST` vers `/api/articles/` est donc bien dirigée vers la fonction 'createArticle'. De même, une requête `GET` vers `/api/articles/test` est dirigée vers 'testApi'.

3.5 Étape 5 : Validation avec Postman

La restructuration est terminée. Le plus important est de vérifier que rien n'est cassé.

1. Assurez-vous que votre serveur est lancé avec 'npm run dev'.
2. Ouvrez Postman et ré-exécutez **toutes** les requêtes de la Semaine 1 :
   - `GET http://localhost:3000/`
   - `GET http://localhost:3000/api/articles/test` (Attention, l'URL a changé !)
   - `POST http://localhost:3000/api/articles`
3. **Toutes les requêtes doivent fonctionner exactement comme avant.** Si c'est le cas, notre refactorisation est un succès ! L'API externe n'a pas changé, mais notre code interne est maintenant propre, organisé et prêt à évoluer.

4 Conclusion et Vision pour la Suite

Félicitations ! Vous avez accompli l'une des étapes les plus importantes pour passer d'un simple script à une application professionnelle. Votre API est maintenant structurée, maintenable et prête à accueillir de nouvelles fonctionnalités sans devenir chaotique.

La semaine prochaine, nous allons enfin nous connecter à une base de données. Nous introduirons **MongoDB** et **Mongoose** pour rendre nos données persistantes et transformer notre API en une application dynamique et réelle.

5 Travail Pratique Complémentaire (À faire par l'étudiant)

Appliquez la même logique de structuration pour une nouvelle ressource : les utilisateurs.

1. **Créez un nouveau contrôleur** : `controllers/userController.js`.
   - Créez et exportez une fonction `getAllUsers` qui renvoie un tableau JSON d'utilisateurs factices.
   - Créez et exportez une fonction `createUser` qui renvoie un message de succès avec les données de 'req.body' (similaire à 'createArticle').
2. **Créez un nouveau routeur** : `routes/userRoutes.js`.
   - Définissez une route `GET /` qui utilise `getAllUsers`.
   - Définissez une route `POST /` qui utilise `createUser`.
3. **Connectez le nouveau routeur** dans 'server.js'. L'application devra l'utiliser pour toutes les requêtes commençant par `/api/users`.
4. **Testez vos nouvelles routes avec Postman** :
   - `GET http://localhost:3000/api/users`
   - `POST http://localhost:3000/api/users` (avec un corps JSON approprié).

Note Importante : Travail à Rendre

Le compte rendu de cette semaine est crucial.

Il doit non seulement décrire les étapes de la refactorisation, mais aussi expliquer pourquoi cette nouvelle structure est meilleure que la précédente. Échéance : La soumission doit

se faire au plus tard la veille de la prochaine séance, à 23h59 précises. Ce compte rendu est obligatoire et noté.