API de Gestion des Tâches

Application Node.js + Express + MongoDB

Réalisé par : OULAID Abdelmounaim

https://github.com/abdelmounaimoulaid/gestions-des-taches

Structure des Fichiers du Projet

Le projet suit une architecture MVC claire avec séparation des responsabilités. Tous les fichiers sources sont organisés dans le dossier `src/` pour une meilleure maintenabilité.

Arborescence du Projet

```
tasks/
├── server.js                 # Point d'entrée principal
├── package.json              # Dépendances npm
├── .env                      # Variables d'environnement
├── .gitignore                # Fichiers exclus de Git
├── README.md                 # Documentation
└── src/
    ├── app.js                # Configuration Express
    ├── models/               # Schémas MongoDB
    │   ├── tasksModel.js     # Modèle Tâches
    │   └── usersModel.js     # Modèle Utilisateurs
    ├── controllers/          # Logique métier
    │   ├── taskController.js # CRUD Tâches
    │   └── userController.js # Auth Utilisateurs
    ├── routes/               # Définition des endpoints
    │   ├── tasksRoute.js     # Routes /api/tasks
    │   └── usersRoute.js     # Routes /api/auth
    └── middlewares/          # Middlewares personnalisés
        └── auth.js           # Vérification JWT
```

Description des Dossiers

- **server.js** : Démarre le serveur HTTP et écoute sur le port configuré
- **src/app.js** : Configure Express, les middlewares de sécurité, et la connexion MongoDB
- **src/models/** : Définit les schémas Mongoose pour la structure des données

- **src/controllers/** : Contient la logique métier pour chaque ressource (Tasks, Users)
- **src/routes/** : Mappe les URLs HTTP aux fonctions des contrôleurs
- **src/middlewares/** : Middlewares personnalisés (authentification JWT)

Flux de Requête

1. **server.js** reçoit la requête HTTP
2. **app.js** applique les middlewares de sécurité (Helmet, CORS, Rate Limiting)
3. **routes/** identifie le contrôleur à appeler
4. **middlewares/auth.js** vérifie le token JWT (si route protégée)
5. **controllers/** exécute la logique métier
6. **models/** interagit avec MongoDB via Mongoose
7. Réponse JSON renvoyée au client

Vue d'ensemble du Projet

Description

Ce projet consiste en la création d'une API RESTful robuste pour la gestion de tâches (To-Do List). L'application permet aux utilisateurs de s'inscrire, de s'authentifier de manière sécurisée et de gérer leurs tâches personnelles (création, lecture, mise à jour, suppression) à travers des endpoints protégés.

L'accent a été mis sur la sécurité (protection contre les injections, XSS, headers HTTP sécurisés) et la performance (architecture non-bloquante de Node.js).

Technologies Utilisées

Node.js | Express.js | MongoDB | Mongoose | JWT (JSON Web Tokens)

Architecture MVC

Le projet suit strictement le modèle architectural **Modèle-Vue-Contrôleur (MVC)** adapté pour une API (sans la partie "Vue" HTML) :

Routes

Dirigent les requêtes HTTP entrantes (GET, POST, PUT, DELETE) vers les méthodes de contrôleur appropriées.

↓

Controllers (Contrôleurs)

Contiennent la logique métier. Ils reçoivent les requêtes, traitent les données, appellent les modèles et renvoient les réponses formatées au client.

↓

Models (Modèles)

Définissent la structure des données (Schémas Mongoose) pour les Utilisateurs et les Tâches. Ils gèrent les interactions directes avec la base de données MongoDB.

↓

MongoDB Database

Point d'Entrée & Configuration

Initialisation - server.js

Ce fichier est le point de départ de l'exécution du serveur. Il charge les variables d'environnement, importe l'application Express configurée, crée le serveur HTTP et démarre l'écoute sur le port défini.

server.js

```
require("dotenv").config();
const http = require("http");
const app = require("./src/app");

app.set('port', process.env.PORT);

const server = http.createServer(app);

server.listen(app.get('port'));
```

Configuration de l'Application - app.js

Le fichier `app.js` centralise la configuration complète d'Express avec une architecture de sécurité robuste incluant Helmet, CORS, protection contre les injections NoSQL/XSS, rate limiting, et validation des variables d'environnement.

src/app.js

```
// 0.imports
const express = require("express");
const helmet = require("helmet");
const cors = require("cors");
const mongoSanitize = require("express-mongo-
sanitize");
const xss = require("xss");
const rateLimit = require("express-rate-limit");
const mongoose = require("mongoose");
const tasksRouter = require("./routes/tasksRoute");
const usersRoute = require("./routes/usersRoute");

// 1.initialisation de express
const app = express();
```

```
// 2.helmet with enhanced security configuration
app.use(helmet({
    contentSecurityPolicy: {
        directives: {
            defaultSrc: ["'self'"],
            styleSrc: ["'self'", "'unsafe-inline'"],
            scriptSrc: ["'self'"],
            imgSrc: ["'self'", "data:", "https:"],
        },
    },
    hsts: {
        maxAge: 31536000,
        includeSubDomains: true,
        preload: true
    }
}));

// 3.cors
app.use(cors());

// 4.body Parsers
app.use(express.json());

// 5.Nettoyage global des données entrantes (NoSQL
Injection)
app.use((req, res, next) => {
    const { sanitize } = mongoSanitize;

    if (req.params) sanitize(req.params);
    if (req.query) sanitize(req.query);
    if (req.body) sanitize(req.body);

    next();
})

// Protection XSS
app.use((req, res, next) => {
    const sanitize = (obj) => {
        for (const key in obj) {
            if (typeof obj[key] === "string") obj[key]
= xss(obj[key])
        }
    }

    if (req.query) sanitize(req.query)
    if (req.params) sanitize(req.params)
    if (req.body) sanitize(req.body)
    next();
})
```

```
// 6.Limitation des requêtes (Rate Limiting)
const globalRateLimit = rateLimit({
    windowMs: 15 * 60 * 1000, // 15min
    max: 100, // 100 requests per 15 minutes
    message: "Too many requests from this IP, please
try again later.",
    standardHeaders: true,
    legacyHeaders: false,
});

app.use(globalRateLimit);

const loginRateLimit = rateLimit({
    windowMs: 15 * 60 * 1000, // 15min
    max: 5
});

// 7.Validate critical environment variables
if (!process.env.MONGO_URL) {
    console.error("FATAL ERROR: MONGO_URL environment
variable is not defined.");
    process.exit(1);
}

if (!process.env.JWT_SECRET) {
    console.error("FATAL ERROR: JWT_SECRET environment
variable is not defined.");
    process.exit(1);
}

// 8.Connexion à MongoDB
mongoose.connect(process.env.MONGO_URL).then((success)
=> {
    console.log("Mongo DB Connected Successfully !")
}).catch((error) => {
    console.log("Error while connecting the MongoDb:
", error)
})

//9. Gestion des routes
app.use("/api/tasks", tasksRouter);
app.use('/api/auth', loginRateLimit, usersRoute);

// 10. 404 Handler
app.use((req, res, next) => {
    res.status(404).json({ message: 'Sorry, the page
you are looking for doesnt exist !' });
});
```

```
module.exports = app;
```

Middlewares de Sécurité Implémentés

- **Helmet avec CSP** : Configuration avancée de Content Security Policy, HSTS avec preload pour forcer HTTPS
- **CORS** : Gestion des requêtes cross-origin
- **NoSQL Injection Protection** : Sanitization automatique de tous les paramètres, query strings et body avec express-mongo-sanitize
- **XSS Protection** : Nettoyage des chaînes de caractères pour bloquer les scripts malveillants
- **Rate Limiting Global** : 100 requêtes max par 15 minutes par IP
- **Rate Limiting Login** : 5 tentatives max par 15 minutes pour prévenir le brute force
- **Environment Validation** : Vérification des variables critiques au démarrage, arrêt si manquantes
- **404 Handler** : Gestion élégante des routes inexistantes

Modèles de Données (Mongoose)

Schéma Tâches

Ce modèle définit la structure stricte d'une tâche avec validation des données.

L'utilisation de l'option `enum` sur le champ **status** garantit que seules les valeurs valides ("À faire", "En cours", "Terminé") sont acceptées. L'option `timestamps: true` génère automatiquement les champs `createdAt` et `updatedAt`.

src/models/tasksModel.js

```
const mongoose = require("mongoose");

const taskStatus = ["À faire", "En cours", "Terminé"];

const tasksSchema = mongoose.Schema(
    {
        titre: { type: String, required: true },
        description: { type: String, required: false
},
        status: {
            type: String,
            enum: taskStatus,
            default: taskStatus[0]
        }
    },
    {
        timestamps: true
    }
);

module.exports = mongoose.model('tasks', tasksSchema);
```

Schéma Utilisateur

Modèle simple pour l'authentification avec validation d'unicité de l'email.

Le plugin `mongoose-unique-validator` est essentiel : il assure qu'il est impossible de créer deux comptes avec la même adresse email, renvoyant une erreur claire avant même de tenter l'insertion en base.

src/models/usersModel.js

```
const mongoose = require("mongoose");
const uniqueValidator = require("mongoose-unique-
validator");

const usersSchema = mongoose.Schema({
    email: { type: String, required: true, unique:
true },
    password: { type: String, required: true }
})

usersSchema.plugin(uniqueValidator)

module.exports = mongoose.model('users', usersSchema);
```

Sécurité & Authentification

Middleware JWT

Ce middleware est le gardien des routes protégées. Il intercepte chaque requête vers l'API des tâches et vérifie l'authentification avant d'autoriser l'accès aux ressources.

src/middlewares/auth.js

```
const jwt = require("jsonwebtoken");
const SECRET_TOKEN = process.env.SECRET_TOKEN;

module.exports = (req, res, next) => {
    try {
        const token =
req.headers.authorization.split(" ")[1];
        const decodeToken = jwt.verify(token,
SECRET_TOKEN);
        const userId = decodeToken.userId;

        req.auth = {
            userId: userId
        };
        next();
    } catch {
        res.status(403).json({ message:
"unauthorized"})
    }
}
```

Fonctionnement

1. Extrait le token du header `Authorization` (Format "Bearer TOKEN")
2. Vérifie la validité du token avec `jwt.verify` et la clé secrète
3. Si valide, attache l'ID utilisateur à l'objet `req` pour les contrôleurs suivants
4. Si invalide ou absent, renvoie une erreur 403 Forbidden

Contrôleur Authentification - Login

Cette fonction gère l'authentification des utilisateurs en vérifiant les identifiants et en générant un token JWT pour les sessions authentifiées.

src/controllers/userController.js (Login)

```
exports.login = (req, res, next) => {
    Users.findOne({ email: req.body.email
}).then((user) => {
        if (!user) res.status(404).json({ message:
"Error !" });

        bycript.compare(req.body.password,
user.password)
            .then((valid) => {
                if (!valid) res.status(404).json({
message: "Error !" });

                res.status(201).json({
                    userId: user._id,
                    token: jwt.sign(
                        { email: user.email },
                        SECRET_TOKEN,
                        { expiresIn: '24h' }
                    )
                })
            });
    })
}
```

Fonctionnement

1. Recherche l'utilisateur dans la base de données par email
2. Compare le mot de passe fourni avec le hash stocké (bcrypt)
3. Si valide, génère un token JWT avec expiration de 24h
4. Retourne le token et l'ID utilisateur pour les requêtes futures

Contrôleur des Tâches

Opérations CRUD

Ce contrôleur implémente les opérations fondamentales de l'API pour la gestion complète des tâches (Create, Read, Update, Delete).

src/controllers/taskController.js

```
const Tasks = require("../models/tasksModel");

exports.getTasks = (req, res, next) => {
    Tasks.find().then((tasks) => {
        res.status(200).json(tasks);
    }).catch((error) => {
        res.status(400).json({error});
    })
}

exports.addTask = (req, res, next) => {
    delete req.body._id;
    const task = new Tasks({
        ...req.body
    });

    task.save().then(() => {
        res.status(201).json({message: "added
Successfully !"});
    }).catch((error) => {
        res.status(400).json({ error });
    })
}

exports.updateTask = (req, res, next) => {
    Tasks.updateOne({ _id: req.params.id }, {
...req.body })
        .then(() => {
            res.status(201).json({message: "updated
!"});
        })
        .catch((error) => {
            res.status(400).json({ error });
        })
}

exports.removeTask = (req, res, next) => {
```

```
    Tasks.deleteOne({ _id: req.params.id }).then(() =>
{
        res.status(204).json({ message: "removed !"
});
    })
    .catch((error) => {
        res.status(400).json({ error });
    })
}
```

Fonctionnement

- **getTasks** : Récupère la liste complète des tâches via `find()`
- **addTask** : Crée une nouvelle tâche. Supprime `_id` par sécurité pour laisser MongoDB gérer l'indexation
- **updateTask** : Modifie une tâche existante en utilisant l'ID passé en paramètre URL
- **removeTask** : Supprime une tâche en ciblant le document par son ID

Définition des Routes

Routage Tâches - Routes Protégées

Ce fichier associe les URLs HTTP aux fonctions du contrôleur. Toutes les routes sont protégées par le middleware d'authentification JWT.

src/routes/tasksRoute.js

```
const express = require("express");
const authMiddleware = require("../middlewares/auth");
const tasksController =
require("../controllers/taskController");

const router = express.Router();

// Routes protégées par authMiddleware
router.get('/', authMiddleware,
tasksController.getTasks);
router.get('/:id', authMiddleware,
tasksController.getOneTask);
router.post('/', authMiddleware,
tasksController.addTask);
router.put('/:id', authMiddleware,
tasksController.updateTask);
router.delete('/:id', authMiddleware,
tasksController.removeTask);

module.exports = router;
```

Important

Le point crucial est l'injection du `authMiddleware` en deuxième argument de chaque route. Cela signifie que **toutes** ces routes sont verrouillées : si le middleware ne valide pas le token, le contrôleur ne sera jamais exécuté.

Routage Authentification - Routes Publiques

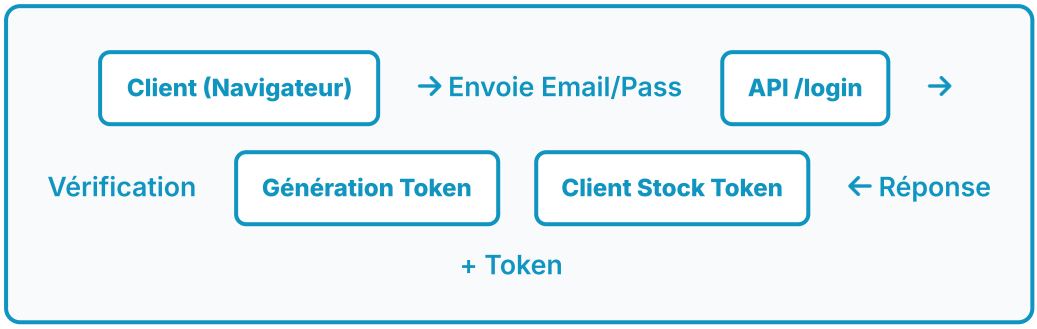
Contrairement aux tâches, ces routes sont publiques (pas de middleware d'authentification). C'est logique car un utilisateur doit pouvoir s'inscrire ou se connecter sans avoir de token au préalable.

src/routes/usersRoute.js

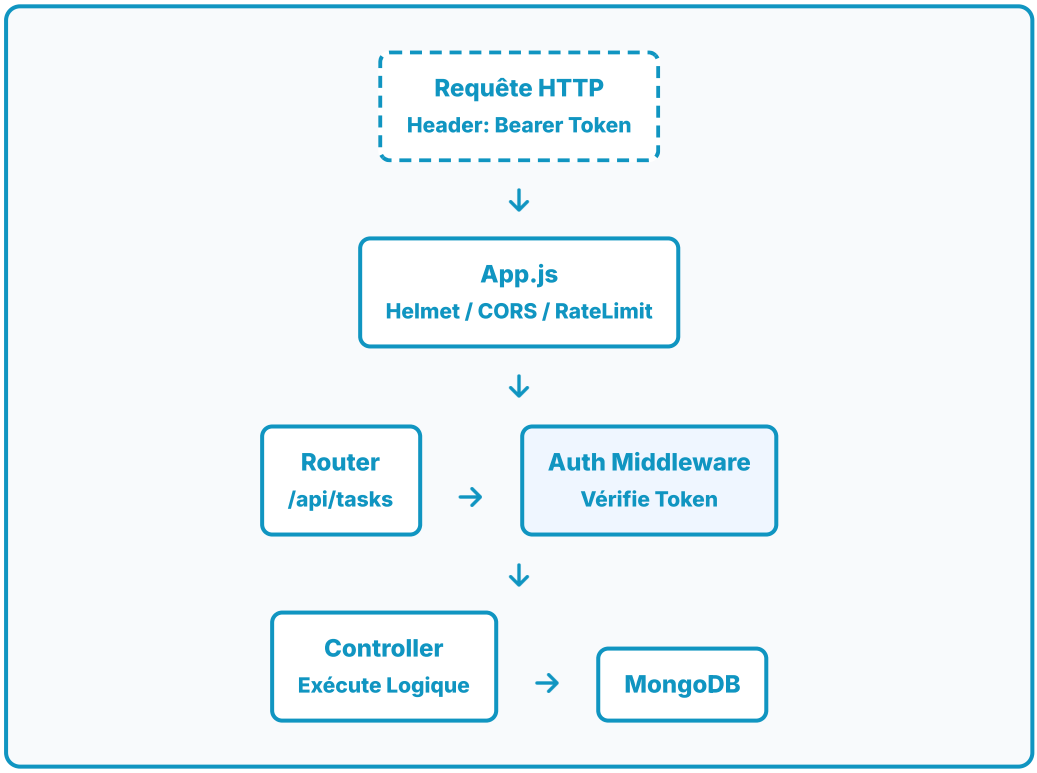
```
const express = require("express");
const userController =
require("../controllers/userController");

const router = express.Router();

// Routes publiques
router.post('/login', userController.login);
router.post('/signup', userController.signUp);

module.exports = router;
```

Architecture & Flux de Données

Flux d'Authentification JWT



Cycle de Vie d'une Requête Sécurisée



Guide d'Utilisation de l'API

Endpoints Publics

Méthode	Endpoint	Description
POST	/api/auth/signup	Création d'un nouveau compte utilisateur
POST	/api/auth/login	Connexion et obtention du JWT

Endpoints Tâches (Authentification Requise)

Header requis : Authorization: Bearer <votre_token>

Méthode	Endpoint	Description
GET	/api/tasks	Récupérer toutes les tâches
GET	/api/tasks/:id	Récupérer une tâche par ID
POST	/api/tasks	Créer une nouvelle tâche
PUT	/api/tasks/:id	Mettre à jour une tâche
DELETE	/api/tasks/:id	Supprimer une tâche

Exemple de format JSON (Tâche)

Format JSON - Tâche

```
{
    "_id": "65a45b9 ... ",
    "titre": "Préparer la documentation",
    "description": "Détailler les endpoints API",
    "status": "En cours",
    "createdAt": "2026-01-15T10:00:00.000Z",
    "updatedAt": "2026-01-15T10:30:00.000Z"
}
```