

# Chapitre 4

## Gestion des erreurs et sécurité

# Gestion des erreurs et sécurité

- Quand on développe une API en Node.js **sans framework**, c'est à nous de gérer
  - Sécurité (JSON malformé, injections, ...)
  - Validation (données incorrectes)
  - Robustesse (crash, erreurs quelconques)
- Ne jamais faire confiance aux données reçues
- Ne jamais laisser un utilisateur faire planter l'application

# Validation des entrées

- Exemple : on vérifie qu'un user contient au moins un name
- Protège contre les champs manquants, JSON corrompu

```
Uploaded using RayThis Extension

req.on('end', () => {
  try {
    const user = JSON.parse(body);

    if (!user.name || typeof user.name !== 'string') {
      res.writeHead(400);
      return res.end(JSON.stringify({ error: 'Le champ "name" est requis' }));
    }

    const created = service.addUser(user);
    res.writeHead(201);
    res.end(JSON.stringify(created));

  } catch (err) {
    res.writeHead(400);
    res.end(JSON.stringify({ error: 'JSON invalide' }));
  }
});
```

# Empêcher les crash

- Dans server.js, on encapsule le routeur pour éviter les erreurs non catchées

```
const server = http.createServer((req, res) => {
  try {
    router(req, res);
  } catch (err) {
    console.error('Erreur interne : ', err);
    res.writeHead(500);
    res.end(JSON.stringify({ error: 'Erreur interne du serveur' }));
  }
});
```

# Sécurisation des accès aux fichiers

- Ajout de try/catch dans les fonctions fs

```
function readUsers() {
  try {
    return JSON.parse(fs.readFileSync(dataPath));
  } catch {
    return []; // fallback au lieu de faire planter l'app
  }
}
```

# Protection contre les gros payloads (limiter taille JSON)

- Évite une attaque qui fait saturer la RAM par exemple



Uploaded using RayThis Extension

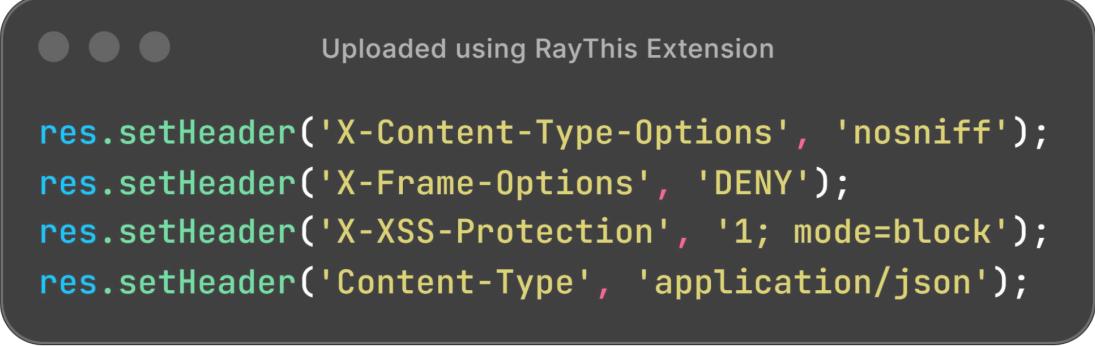
```
let body = '';
const MAX_SIZE = 1e6; // 1MB

req.on('data', chunk => {
  body += chunk;

  if (body.length > MAX_SIZE) {
    res.writeHead(413); // Payload Too Large
    res.end(JSON.stringify({ error: 'Payload trop volumineux' }));
    req.destroy();
  }
});
```

# Ajout des headers “classiques” de sécurité

- Empêcher des attaques navigateurs



Uploaded using RayThis Extension

```
res.setHeader('X-Content-Type-Options', 'nosniff');
res.setHeader('X-Frame-Options', 'DENY');
res.setHeader('X-XSS-Protection', '1; mode=block');
res.setHeader('Content-Type', 'application/json');
```

# Ajout d'un service logger

- A utiliser après chaque catch() au moins



Uploaded using RayThis Extension

```
function logError(err) {
    console.error(`[${new Date().toISOString()}] ERROR:`, err);
}
```

# **Exercices**

# Chapitre 5

# EJS Templates

# Introduction à EJS et configuration

- Un moteur de template utilise un langage différent pour définir un template HTML
- EJS pour Embedded Javascript Templates (côté client et serveur) est un moteur de template
- <https://ejs.co/>

```
...                               Uploaded using RayThis Extension

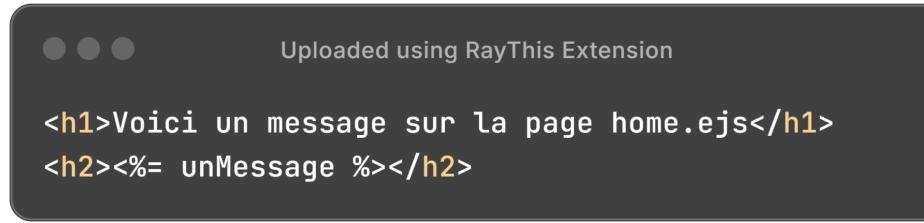
app.set('views', './views');
app.set('view engine', 'ejs');

let message = "Hello World!";

app.get('/', (req, res) => {
  res.render('home', { unMessage:message });
});

// On récupère les fichiers statiques dans le dossier public
app.use(express.static('public'));
```

# Une page EJS dans /views/home.ejs



# Layout EJS

● ● ● Uploaded using RayThis Extension

```
<%- include('_header') %>
<body>
  <%-include ("_product_form")%>
</body>
<%-include ("_footer")%>
```

# Les différentes balises avec EJS

```
<%= 'Scriptlet' %> // Affiche une variable
<%- 'Scriptlet non échappé' %> // Affiche une variable sans échapper les caractères HTML
<% if (condition) { %> // Exécute du code conditionnel
    <p>Condition vraie</p>
<% } else { %> // Exécute du code si la condition est fausse
    <p>Condition fausse</p>
<% } %> // Fin du bloc conditionnel
<% for (let i = 0; i < array.length; i++) { %> // Boucle sur un tableau
    <p>Élément: <%= array[i] %></p>
<% } %> // Fin de la boucle
```

# **Exercices**



# TP 3 : Validation des acquis