Design Patterns :
Développement Web

Julien Provillard

PRÉSENTATION DU MODULE EXPRESS

Le module express

```
import express from "express";
import { createServer } from "http";

// app est la fonction de rappel créée par Express
const app = express();

// paramétrage de app (à venir)
// ...

const server = createServer(app);
const port = 3000;

// lancement du serveur avec log du moment où il est prêt
server.listen(port, () => console.log(`Server launched on port ${port}`));
```

Il faut correctement paramétrer la dépendance dans `package.json` puis réaliser l'installation avec `npm`.

Création de la fonction de rappel du serveur.

Utilisation classique via le module `http` de node.

Le module express

```
app.all("*", mainRouter);

function mainRouter(req, res) {
  // alternative à res.writeHead(200, { "Conte
  res.statusCode = 200;
  res.setHeader("Content-Type", "text/html; charset=utf-8");
  res.write("<h1>Request successful</h1>");
  res.write(`<p>Received a request ${req.method} on ${req.url}.</p>`);
  res.end();
}
```

Installation d'une route pour la requête :

- `.all` pour réagir à toutes les méthodes http
- `"*"` pour réagir à n'importe quelle url
- `mainRouter` est le callback mis en place

Callback classique comme le ferait node.

D'ailleurs, ça ressemble un peu trop à du node.

Le module express

```
app.all("*", mainRouter);

function mainRouter(req, res) {
  let body = `<h1>Request successful</h1>
  <p>Received a request ${req.method} on ${req.url}.</p>`;
  res.status(200);
  // <=> res.set("Content-Type", "text/html; charset=utf-8")
  res.type("html");
  res.send(body);
}
```

On construit le corps de la réponse.

On précise le type du contenu de la réponse. En interne, le module `mime` est utilisé.

La méthode `send` de `express` combine les méthodes `write` et `end` de node.

Le module express

```
app.all("*", mainRouter);

function mainRouter(req, res) {
  let body = `<h1>Request successful</h1>
  <p>Received a request ${req.method} on ${req.url}.</p>`;
  res.status(200).type("html").send(body);
}
```

L'API de express autorise le chaînage pour simplifier les appels successifs sur le même objet.

Le module express

```
app.all("*", mainRouter);

function mainRouter(req, res) {
  let body = `<h1>Request successful</h1>
  <p>Received a request ${req.method} on ${req.url}.</p>`;
  res.send(body);
}
```

Les appels à `status` et `type` ont été supprimés.
Par défaut, `send` répond avec le code 200 et envoie du html (headers automatiques).

Le module express

```
app
  .get("/foo", (req, res) => res.send("<h1>Welcome foo!</h1>"))
  .get("/bar", (req, res) => res.send("<h1>Hi bar!</h1>"))
  .all("*", (req, res) =>
    res.status(404).send("<h1>This page does not exists.</h1>"));
```

- Exemple avec plusieurs routes :
  - La première réagit aux requêtes GET sur l'url /foo.
  - La seconde réagit aux requêtes GET sur l'url /bar.
  - La dernière réagit à toutes les requêtes quels que soient la méthode et l'url.
- Les routes sont analysées successivement. La première qui convient est utilisée.

Le module express

```
app
  .get("/foo", (req, res) => res.send("<h1>Welcome foo!</h1>"))
  .get("/bar", (req, res) => res.send("<h1>Hi bar!</h1>"))
  .all("*", (req, res) =>
    res.status(404).send("<h1>This page does not exists.</h1>"));
```

- L'objet app dispose d'une méthode pour chaque verbe http (get, post, put, delete, …).
- La méthode all s'active quel que soit le verbe http utilisé.
- Il faut préciser l'url pour laquelle on installe la route. De manière générale, c'est une **expression régulière**.

Middlewares

- Objectif : ajouter du logging sur la console.

```
app
  .all("*", (req, res) => console.log(`Request ${req.method} on ${req.url}.`))
  .get("/foo", (req, res) => res.send("<h1>Welcome foo!</h1>"))
  .get("/bar", (req, res) => res.send("<h1>Hi bar!</h1>"))
  .all("*", (req, res) =>
    res.status(404).send("<h1>This page does not exists.</h1>"));
```

- Echec, le serveur ne répond plus. Pourquoi ?
- La première route a déjà intercepté toutes les requêtes. Les routes qui suivent sont ignorées.

Middlewares

- Objectif : ajouter du logging sur la console.

```
app
  .all("*", (req, res, next) => {
    console.log(`Request ${req.method} on ${req.url}.`);
    next(); // au(x) suivant(s)
  })
  // ...
```

- Les callbacks de `express` autorisent trois paramètres : la requête, la réponse et une fonction qui relance l'analyse de la requête.
- Ici, on logue la requête puis on passe la main aux routes suivantes pour gérer la réponse.

Middlewares

- Objectif : ajouter du logging sur la console.

```
app
  .use((req, res, next) => {
    console.log(`Request ${req.method} on ${req.url}.`);
    next(); // au(x) suivant(s)
  })
  // ...
```

- Les fonctions qui ne terminent pas la requête sont des middlewares.
- Les middlewares sont usuellement mis en place avec la méthode `use`.
- On peut préciser une url qui active le middleware si nécessaire. Les appels `.use(url, fn)` et `.all(url, fn).all(url/*, fn)` sont presque équivalents.

Gestion des erreurs

- Express permet de définir des middleware spéciaux pour la gestion d'erreur.
- Un gestionnaire d'erreurs doit avoir **quatre arguments explicites** et être installé avec use.

```
app
  // ...
  .use((err, req, res, next) => // next obligatoire même si non utilisé
    res.status(err.status || 500).send(`<h1>${err.message}</h1>`));
```

- Toutes les erreurs sur l'url d'installation sont capturées pour être traitées.
- Express utilise un gestionnaire par défaut en dernier traitement.

Gestion des erreurs

- Un gestionnaire d'erreurs ne s'active que pour les erreurs synchrones.

```
app
  .all("*", () => { throw new Error(); }) // le gestionnaire s'active
  .use((err, req, res, next) =>
    res.status(err.status || 500).send(`<h1>${err.message}</h1>`));
```

- Dans le code asynchrone, les erreurs ne sont pas capturées.

```
app
  .all("*", () => setTimeout(() => { throw new Error(); })) // plantage serveur
  .use((err, req, res, next) =>
    res.status(err.status || 500).send(`<h1>${err.message}</h1>`)
  );
```

Gestion des erreurs

- Il est possible d'appeler la méthode `next` avec une erreur en argument pour l'envoyer directement au gestionnaire d'erreurs.

```
app
  .all("*", (req, res, next) => next(new Error())) // le gestionnaire s'active
  .use((err, req, res, next) =>
    res.status(err.status || 500).send(`<h1>${err.message}</h1>`));
```

```
app
  .all("*", (req, res, next) => setTimeout(() => next(new Error()))) // ici aussi
  .use((err, req, res, next) =>
    res.status(err.status || 500).send(`<h1>${err.message}</h1>`));
```

Exemple

```
import express from "express";
import createError from "http-errors";
import morgan from "morgan";
import { fileURLToPath } from "url";

const app = express();
const publicPath = fileURLToPath(new URL("./public", import.meta.url));

app
  .use(morgan("dev"))
  .use(express.static(publicPath))
  .use(express.json())
  .use(express.urlencoded({ extended: false }))
  .use((req, res, next) => { console.log("query =", req.query); next(); }
  .use((req, res, next) => { console.log("body =", req.body); next(); }
  // ... autres middlewares et routes
  .use((req, res, next) => next(createError(404)))
  .use((err, req, res, next) => res.send(`<h1>${err.message}</h1>`));
```

Un logger, dev est une configuration prédéfinie.

Exemple

```
import express from "express";
import createError from "http-errors";
import morgan from "morgan";
import { fileURLToPath } from "url";

const app = express();
const publicPath = fileURLToPath(new URL("./public", import.meta.url));

app
  .use(morgan("dev"))
  .use(express.static(publicPath))
  .use(express.json())
  .use(express.urlencoded({ extended: false }))
  .use((req, res, next) => { console.log("query =", req.query); next(); }
  .use((req, res, next) => { console.log("body =", req.body); next(); }
  // ... autres middlewares et routes
  .use((req, res, next) => next(createError(404)))
  .use((err, req, res, next) => res.send(`<h1>${err.message}</h1>`));
```

Création du chemin local, `./public` par rapport à `import.meta.url` (l'url du fichier courant).

Un serveur de fichier statique sur un chemin local. Le middleware passe la main si le fichier n'est pas trouvé.

Exemple

```
import express from "express";
import createError from "http-errors";
import morgan from "morgan";
import { fileURLToPath } from "url";

const app = express();
const publicPath = fileURLToPath(new URL("./public", import.meta.url));

app
  .use(morgan("dev"))
  .use(express.static(publicPath))
  .use(express.json())
  .use(express.urlencoded({ extended: false }))
  .use((req, res, next) => { console.log("query =", req.query); next(); }
  .use((req, res, next) => { console.log("body =", req.body); next(); }
  // ... autres middlewares et routes
  .use((req, res, next) => next(createError(404)))
  .use((err, req, res, next) => res.send(`<h1>${err.message}</h1>`));
```

Un parser pour le corps des requêtes POST dont le type est `application/json`.
Peuple `req.body` en utilisant `JSON.parse`.

Exemple

```
import express from "express";
import createError from "http-errors";
import morgan from "morgan";
import { fileURLToPath } from "url";

const app = express();
const publicPath = fileURLToPath(new URL("./public", import.meta.url));

app
  .use(morgan("dev"))
  .use(express.static(publicPath))
  .use(express.json())
  .use(express.urlencoded({ extended: false }))
  .use((req, res, next) => { console.log("query =", req.query); next(); }
  .use((req, res, next) => { console.log("body =", req.body); next(); }
  // ... autres middlewares et routes
  .use((req, res, next) => next(createError(404)))
  .use((err, req, res, next) => res.send(`<h1>${err.message}</h1>`));
```

Un parser pour le corps des requêtes POST dont le type est `application/x-www-form-urlencoded`.
C'est le type renvoyé par les formulaire.
Peuple `req.body`.

Exemple

```
import express from "express";
import createError from "http-errors";
import morgan from "morgan";
import { fileURLToPath } from "url";

const app = express();
const publicPath = fileURLToPath(new URL("./public", import.meta.url));

app
  .use(morgan("dev"))
  .use(express.static(publicPath))
  .use(express.json())
  .use(express.urlencoded({ extended: false }))
  .use((req, res, next) => { console.log("query =", req.query); next(); }
  .use((req, res, next) => { console.log("body =", req.body); next(); }
  // ... autres middlewares et routes
  .use((req, res, next) => next(createError(404)))
  .use((err, req, res, next) => res.send(`<h1>${err.message}</h1>`));
```

La chaîne de requête représentée dans un objet.
L'imbrication est gérée.

Exemple

```
import express from "express";
import createError from "http-errors";
import morgan from "morgan";
import { fileURLToPath } from "url";

const app = express();
const publicPath = fileURLToPath(new URL("./public", import.meta.url));

app
  .use(morgan("dev"))
  .use(express.static(publicPath))
  .use(express.json())
  .use(express.urlencoded({ extended: false }))
  .use((req, res, next) => { console.log("query =", req.query); next(); }
  .use((req, res, next) => { console.log("body =", req.body); next(); }
  // ... autres middlewares et routes
  .use((req, res, next) => next(createError(404)))
  .use((err, req, res, next) => res.send(`<h1>${err.message}</h1>`));
```

Le résultat des middlewares express.json et express.urlencoded.
Il existe d'autres middlewares qui peuplent req.body de façons différentes.

Exemple

```
import express from "express";
import createError from "http-errors";
import morgan from "morgan";
import { fileURLToPath } from "url";

const app = express();
const publicPath = fileURLToPath(new URL("./public", import.meta.url));

app
  .use(morgan("dev"))
  .use(express.static(publicPath))
  .use(express.json())
  .use(express.urlencoded({ extended: false }))
  .use((req, res, next) => { console.log("query =", req.query); next(); }
  .use((req, res, next) => { console.log("body =", req.body); next(); }
  // ... autres middlewares et routes
  .use((req, res, next) => next(createError(404)))
  .use((err, req, res, next) => res.send(`<h1>${err.message}</h1>`));
```

Fonction utilitaire pour la création d'erreurs à partir des codes de retour http.

Exemple

```
import express from "express";
import createError from "http-errors";
import morgan from "morgan";
import { fileURLToPath } from "url";

const app = express();
const publicPath = fileURLToPath(new URL("./public", import.meta.url));

app
  .use(morgan("dev"))
  .use(express.static(publicPath))
  .use(express.json())
  .use(express.urlencoded({ extended: false }))
  .use((req, res, next) => { console.log("query =", req.query); next(); }
  .use((req, res, next) => { console.log("body =", req.body); next(); }
  // ... autres middlewares et routes
  .use((req, res, next) => next(createError(404)))
  .use((err, req, res, next) => res.send(`<h1>${err.message}</h1>`));
```

Le gestionnaire d'erreurs.

CRÉATION D’UN SERVEUR

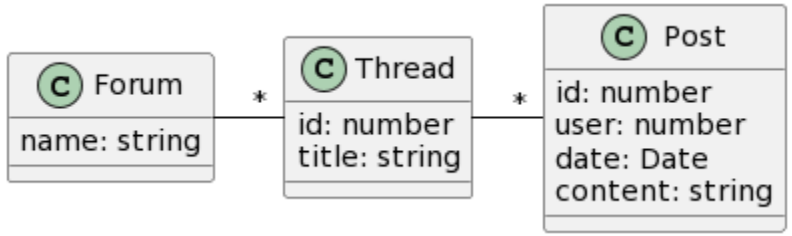
Objectif

- L'objectif est de créer un serveur pour un forum.

- Fonctionnalités sans authentification :
  - Pouvoir consulter les sujets et les messages.
- Fonctionnalités avec authentification :
  - Pouvoir ajouter un sujet.
  - Pouvoir ajouter/supprimer/éditer un message.

Représentation

- Un forum est composé d'un nom et d'une liste de discussions.
- Une discussion est composée d'un identifiant unique, d'un titre et d'une liste de messages.
- Un message est composé d'un identifiant unique, d'un identifiant d'utilisateur (l'auteur du message), d'une date et d'un contenu.



server.js

- C’est le point d’entrée pour le serveur. La partie Express est externalisée.

```
import { createServer } from "http";
import { app } from "./app.js";

// app est la fonction de rappel créée par Express
const server = createServer(app);
const port = process.env.PORT || 3000;

// lancement du serveur avec log du moment où il est prêt
server.listen(port, () => console.log(`Server launched on port ${port}`));
```

app.js

- C'est l'application Express. Les routes sont externalisées.

```
import * as routes from "./routes.js";
export const app = express();

app
  .use(morgan("dev"))
  .use(express.static(fileURLToPath(new URL("./public", import.meta.url))))
  .use(express.json())
  .use(express.urlencoded({ extended: false }))
  .get("/", routes.index)
  .get("/threads/:threadId/", routes.getThread)
  .use((req, res, next) => next(createError(404)))
  .use((err, req, res, next) => {
    res.status(err.status || 500).send(`<h1>${err.message || "Internal error"}</h1>`);
  });
```

Ici, `:threadId` est un paramètre qui capture une partie de l'url et le place dans `req.params.threadId`.
Pour une requête sur `/threads/1/`, on aura donc `req.params.threadId == "1"`.

route.js

- Ce sont les routes. Les vues sont externalisées.

```
import indexView from "./views/index.js";
import threadView from "./views/thread.js";

let currentId = 0;
const forum = createDefaultForum();

function createDefaultForum() { /* ... */ }

function getNewId() { return ++currentId; }

export function index(req, res) { res.send(indexView(forum)); }

export function getThread(req, res, next) {
  let id = req.params.threadId;
  let thread = forum.threads.find((thread) => thread.id == id);
  if (!thread) { next(createError(404)); } else { res.send(threadView(thread)); }
}
```

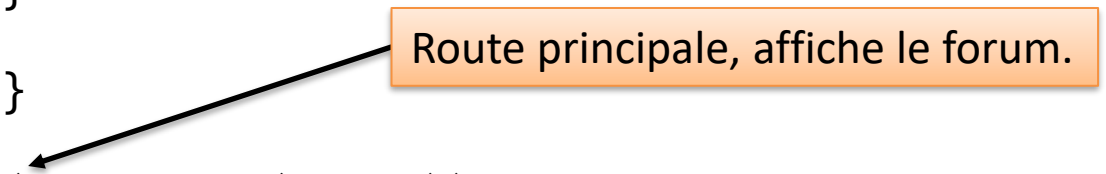
Les données représentant le forum (solution temporaire).

Générateur d'identifiants uniques.
Utilisé dans `createDefaultForum` et servira lors des créations des discussions / messages.

route.js

- Ce sont les routes. Les vues sont externalisées.

```
import indexView from "./views/index.js";
import threadView from "./views/thread.js";

let currentId = 0;
const forum = createDefaultForum();

function createDefaultForum() { /* ... */ }

function getNewId() { return ++currentId; }

export function index(req, res) { res.send(indexView(forum)); }

export function getThread(req, res, next) {
  let id = req.params.threadId;
  let thread = forum.threads.find((thread) => thread.id == id);
  if (!thread) { next(createError(404)); } else { res.send(threadView(thread)); }
}
```



view/index.js

- Affichage du forum, création dynamique de page html.

```
export default function indexView(forum) {
  return `<h1>${forum.name}</h1>
    <h2>Discussions</h2>
    ${buildThreads(forum.threads)}`;
}

function buildThreads(threads) {
  return `<table id=threads><tbody>
    ${threads.map(buildThread).join("")}</tbody></table>`;
}

function buildThread(thread) {
  return `<tr><td><a href=/threads/${thread.id}/>${thread.title}</a></td></tr>`;
}
```

route.js

- Ce sont les routes. Les vues sont externalisées.

```
import indexView from "./views/index.js";
import threadView from "./views/thread.js";

let currentId = 0;
const forum = createDefaultForum();

function createDefaultForum() { /* ... */ }

function getNewId() { return ++currentId; }

export function index(req, res) { res.send(indexView(forum)); }

export function getThread(req, res, next) {
  let id = req.params.threadId;
  let thread = forum.threads.find((thread) => thread.id == id);
  if (!thread) { next(createError(404)); } else { res.send(threadView(thread)); }
}
```

Affichage d'une discussion.
Il faut vérifier que l'identifiant correspond à celui d'une discussion existante avant de demander son affichage.

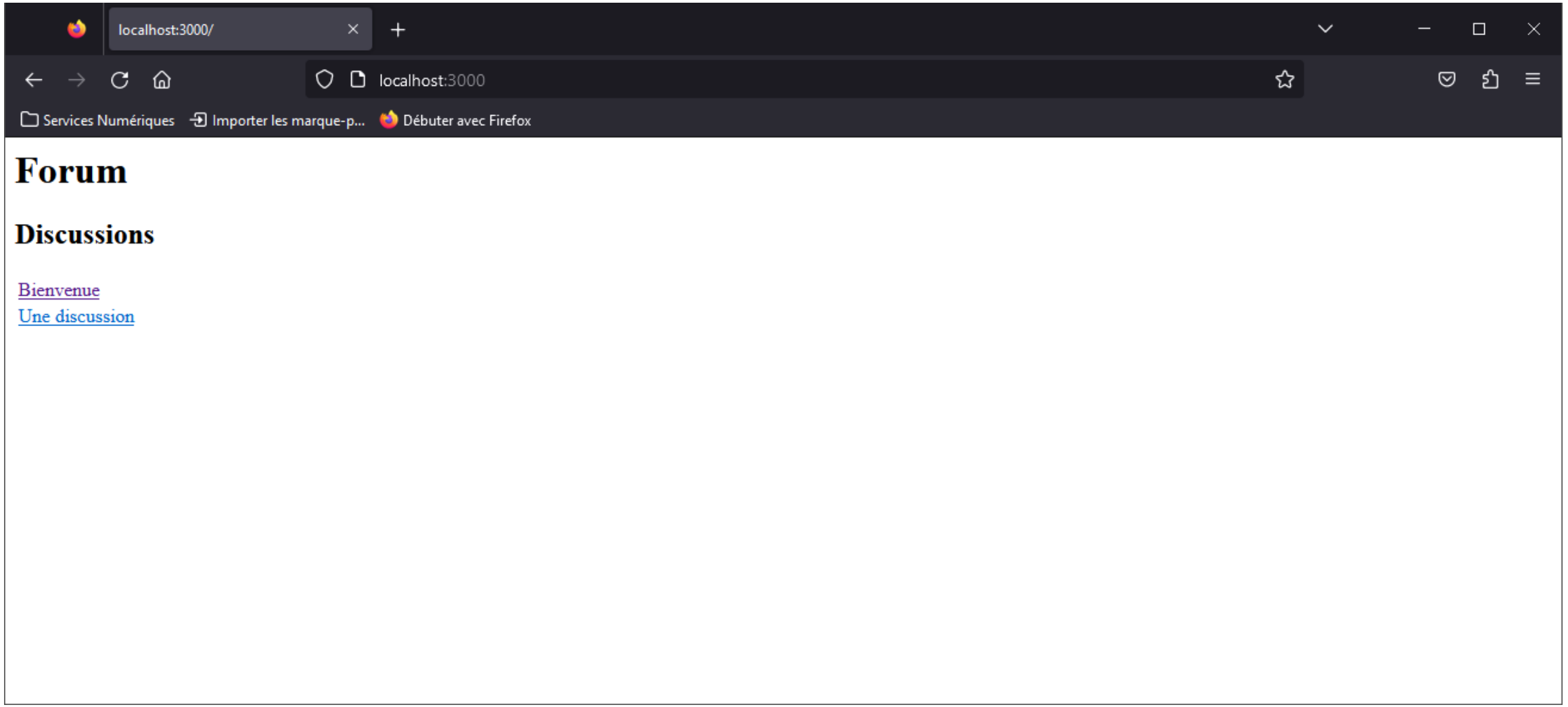
La vue est de nouveau externalisée.

view/thread.js

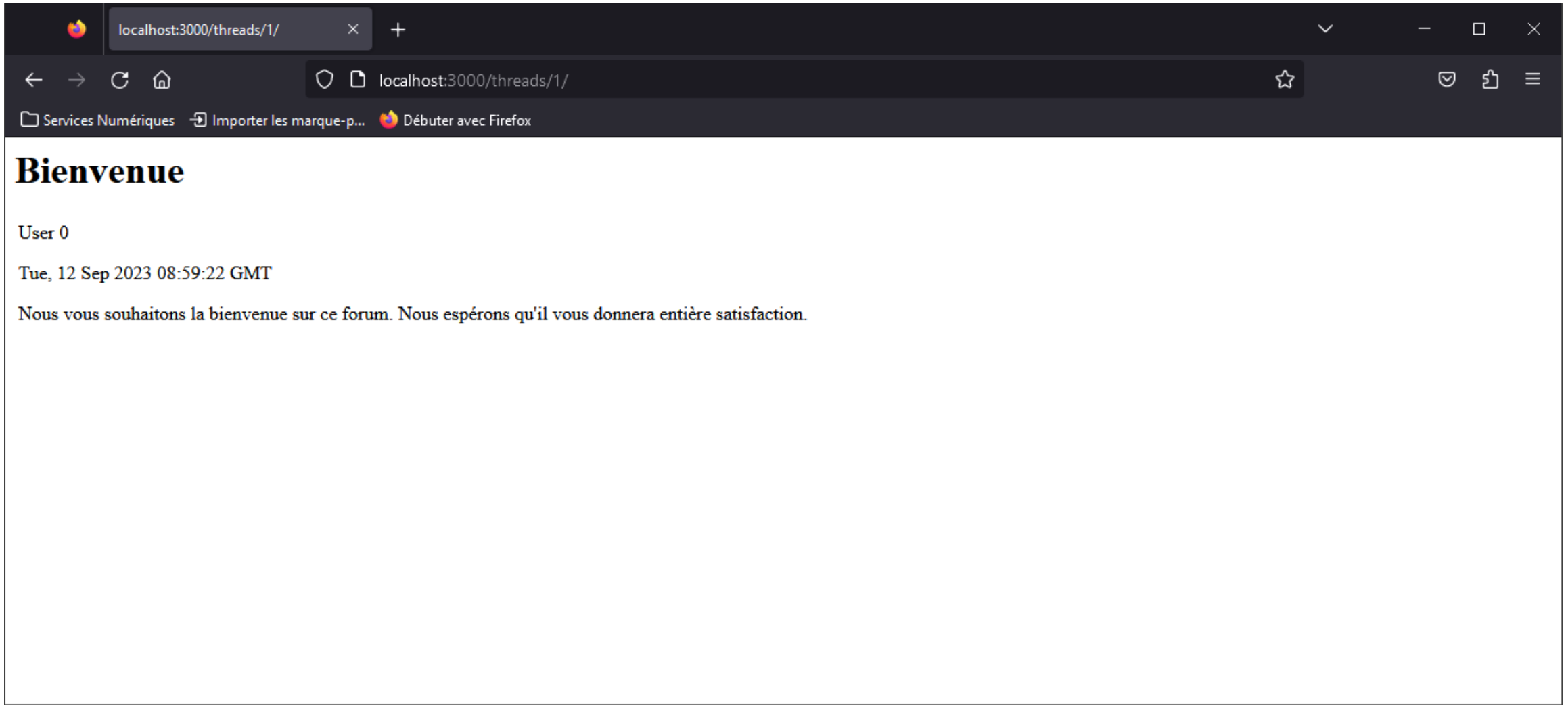
- Affichage d'une discussion, création dynamique de page html.

```
export default function threadView(thread) {
  return `<h1>${thread.title}</h1>
    ${buildPosts(thread.posts)}`;
}

function buildPosts(posts) {
  return `<table id="posts"><tbody>${posts.map(buildPost).join("")}</tbody></table>`;
}

function buildPost(post) {
  return `<tr><td>
    <p class="user">User ${post.user}</p>
    <p class="date">${post.date.toUTCString()}</p>
    <p class="content">${post.content}</p>
  </td></tr>`;
}
```

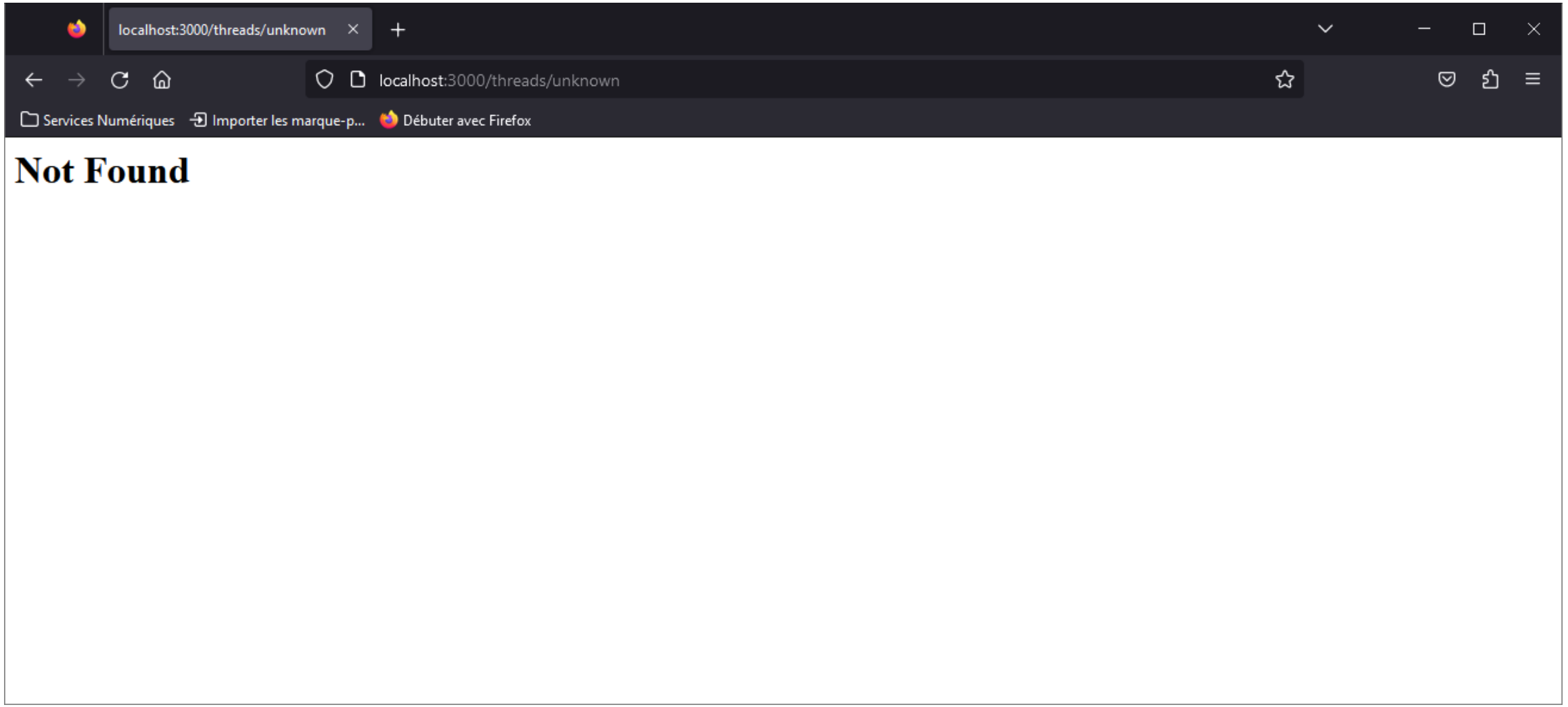
Essai avec le forum de test



Essai avec le forum de test



Essai avec le forum de test



Feedback

- Le serveur fonctionne !
- L'architecture de base est posée. Il faut désormais l'étoffer.

- Quel est le problème le plus pressant ?
- La gestion des vues.
- En l'état, elle passera difficilement à l'échelle.
- Express propose des solutions élégantes pour cela.

View engine

- Express autorise l'utilisation de moteurs de modèles pour envoyer des réponses sous forme de pages html.

- Il y a trois étapes à respecter :
  - Indiquer où sont situés les modèles des vues.

```
app.set("views", fileURLToPath(new URL("./views", import.meta.url)));
```

  - Indiquer le moteur de modèles utilisé.

```
app.set("view engine", "ejs");
```

  - Demander l'envoie de la vue.

```
res.render("someView", data)
```

EJS

- Embedded JavaScript (EJS) est un langage pour le moteur de modèles de Express.
- C’est loin d’être le seul ni forcément le plus populaire.
- Pourquoi le choisir ?
  - Installation et utilisation simples avec Express.
  - Syntaxe proche de html/js, apprentissage aisé.
  - Tremplin vers d’autres langages plus modernes.

EJS

- Basiquement, on fait ce que le nom du langage suggère : écriture d'une page web avec du code javascript intégré.
- Entre des balises <% et %>, on peut écrire du code javascript qui sera interprété. On les utilise pour les structures de contrôle.
- Entre des balises <%= et %>, on peut écrire du code javascript dont le résultat est transformé en chaîne de caractères et inséré dans le document html.
- Entre des balises <%- et %>, on peut inclure du code ejs brut. Cette possibilité sert à créer sa page web en agrégeant des vues partielles.

EJS : Exemple

- On configure le moteur de modèles du serveur pour EJS.

```
app.set("views", fileURLToPath(new URL("./views", import.meta.url)));
app.set("view engine", "ejs");
```

- Les vues seront donc recherchées dans le répertoire `views`.
- Il suffit de router correctement vers ces vues.

```
export function index(req, res) { res.render("index", forum); }

export function getThread(req, res, next) {
  let id = req.params.threadId;
  let thread = forum.threads.find((thread) => thread.id == id);
  if (!thread) { next(createError(404)); } else { res.render("thread", thread)); }
}
```

EJS : Exemple

- On configure le moteur de modèles du serveur pour EJS.

```
app.set("views", fileURLToPath(new URL("./views", import.meta.url)));
app.set("view engine", "ejs");
```

- Les vues seront donc recherchées dans le répertoire `views`.
- Il suffit de router correctement vers ces vues.

On recherche la vue `views/index.ejs`.

```
export function index(req, res) { res.render("index", forum); }

export function getThread(req, res, next) {
  let id = req.params.threadId;
  let thread = forum.threads.find((thread) => thread.id == id);
  if (!thread) { next(createError(404)); } else { res.render("thread", thread)); }
}
```

Les propriétés de `forum` seront accessibles depuis l'objet global dans la vue.

index.ejs

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Index</title>
  </head>
  <body>
    <h1><%= name %></h1>
    <h2>Discussions</h2>
    <table id="threads">
      <% for (thread of threads) { %>
      <%- include("thread-link", thread); %>
      <% } %>
    </table>
  </body>
</html>
```

index.ejs

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Index</title>
  </head>
  <body>
    <h1><%= name %></h1>
    <h2>Discussions</h2>
    <table id="threads">
      <% for (thread of threads) { %>
      <%- include("thread-link", thread); %>
      <% } %>
    </table>
  </body>
</html>
```

Avec la balise <%=, le nom du forum devient le contenu de la balise <h1>.

Boucle javascript classique

Qui permet d'inclure une vue partielle pour chaque discussion. Cet appel revient à inliner le résultat de `render("thread-link", thread)`.

On retrouve le nom du forum et les discussions dans l'objet global.

thread-link.ejs

- On a une vue partielle pour chaque discussion.

```
<tr>
  <td>
    <a href="/threads/<%= id %>"><%= title %></a>
  </td>
</tr>
```

- Ici, il n'était pas vraiment necessaire de séparer les vues.
- Cela sert surtout à montrer le fonctionnement de l'inclusion de vues partielles.

CSS

- Pour ajouter des feuilles de style, il suffit de les déclarer dans les modèles EJS.

```
<head>
  <link rel="stylesheet" href="/stylesheets/index.css" />
  <!-- ... -->
</head>
```

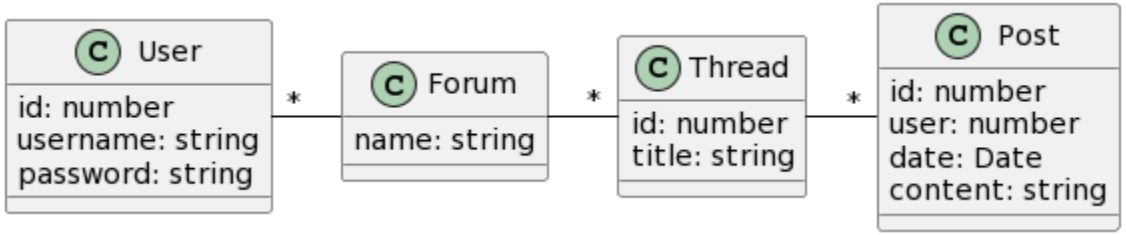
- Au chargement de la page, une requête GET est automatiquement réalisée pour récupérer le fichier `/stylesheets/index.css`.
- Il faut donc le placer dans le dossier `./public` pour qu'il soit servi par `express.static`.
- Ces manipulations ne nécessitent pas de rédémarrage serveur.

Authentification

- Pour les autres opérations qui modifient les données du forum, il faut mettre en place un système d'authentification.
- Il faut pouvoir créer un nouveau compte ou se connecter depuis un compte existant.
- Il faut également décider de la manière dont l'authentification est assurée entre le serveur et le client.
- Normalement, il faudrait réaliser ces opérations via le protocole https qui crypte les échanges.
- Concrètement, la partie `express` est inchangée mais `node` utilise le module `https` au lieu de `http`. Cela nécessite un certificat.

Création de compte

- On fait évoluer le modèle du forum.



- On ajoute deux nouvelles routes pour la création du compte.

```
app
  // ...
  .get("/account/new", routes.getAccountCreationPage)
  .post("/account/new", routes.createAccount)
```

- Ces deux routes ne se distinguent que par la méthode utilisée.

Création de compte

- Pour accéder à la page de création, on utilise le modèle ejs.

```
export function getAccountCreationPage(req, res) { res.render("account"); }
```

- On prévoit déjà de pouvoir passer optionnellement des informations à la vue en cas de problème (nom d'utilisateur déjà pris).
- Dans la vue, pour tester si une propriété data a été définie, il faut vérifier si `locals.data` est `undefined`.
- Si l'on appelle `data` directement alors que la propriété n'est pas définie, une erreur liée à ejs se produit.

`res.locals`

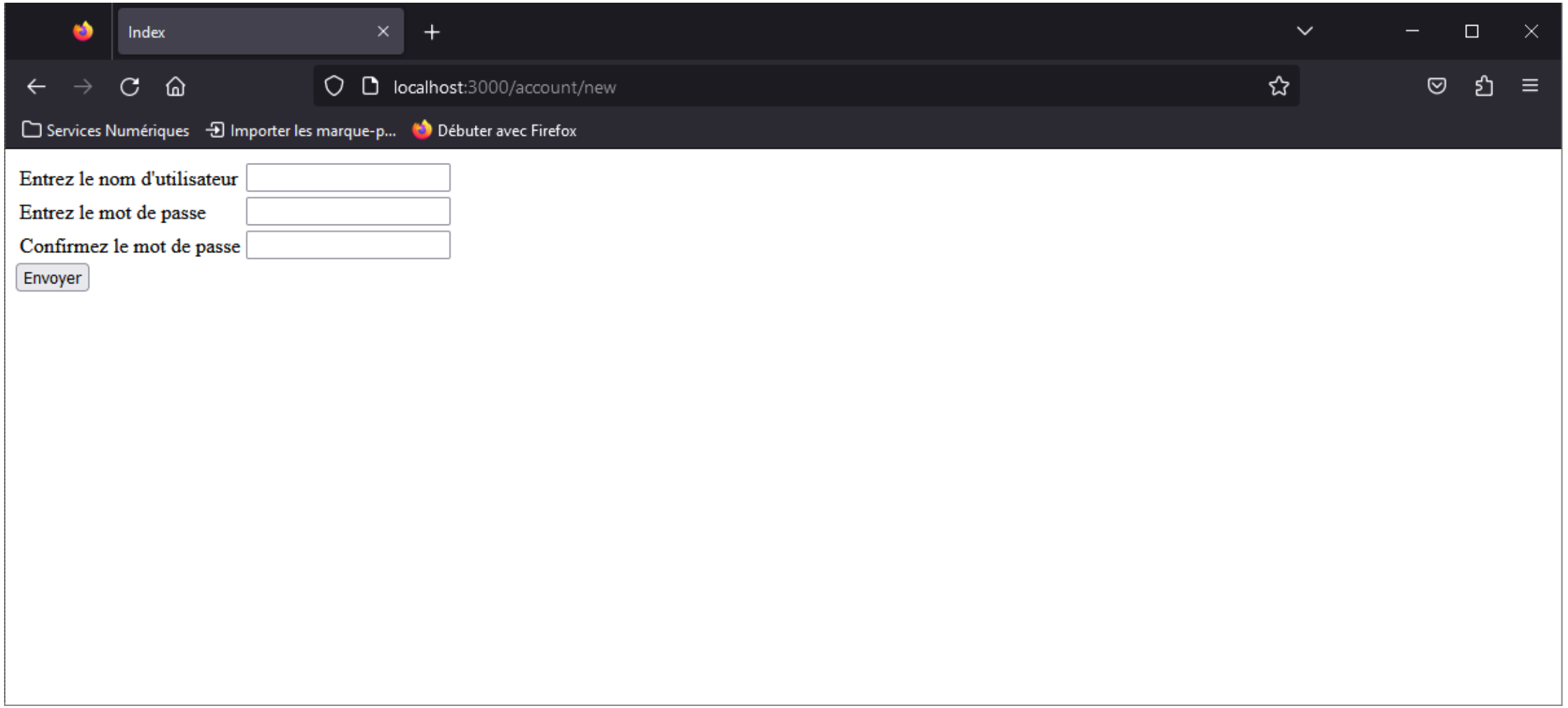
- De manière générale, l'objet `res.locals` (abrégé en `locals` dans les vues) contient toutes les variables locales nécessaires pour générer la réponse.
- Il peut être peuplé manuellement ou automatiquement par certains appels tels que `res.render(...)`.
- Dans ejs, on peut accéder aux variables locales **définies** en utilisant simplement leur nom (`data` vs `locals.data`).
- C'est de cette manière que des middlewares différents peuvent (et devraient) communiquer entre eux.

Le corps de account.ejs

```
<form name="accountCreation" action="/account/new" method="POST">
  <table>
    <tbody>
      <tr><td><label for="username">Entrez le nom d'utilisateur</label></td>
          <td><input name="username" value="<%= locals.username ?? '' %>" /></td>
          <td><label><%= locals.message ?? '' %></label></td></tr>
      <tr><td><label for="password">Entrez le mot de passe</label></td>
          <td><input type="password" name="password" value="" /></td>
          <td><label id="passwordMessage"></label></td></tr>
      <tr><td><label for="passwordRepeated">Confirmez le mot de passe</label></td>
          <td><input type="password" id="passwordRepeated" value="" /></td>
          <td><label id="passwordRepeatedMessage"></label></td></tr>
    </tbody>
  </table>
  <div><input type="submit" /></div>
</form>
```

`id` au lieu de `name` pour éviter que ce champ ne soit transmis à la soumission du formulaire.

Le corps de `account.ejs`



Les scripts de account.ejs

```
<script src="/javascripts/hash.js"></script>
<script>
  let accountForm = document.forms.accountCreation;
  accountForm.addEventListener("submit", (e) => {
    e.preventDefault();
    validateAccountCreation();
  });

  async function validateAccountCreation() {
    // ...
  }
</script>
```

Script qui fournit la fonction de hachage `hashSHA256`.

On annule toutes les soumissions initiées par l'utilisateur pour procéder à des vérifications.

Les scripts de account.ejs

```
async function validateAccountCreation() {
  let passwordMessage = document.getElementById("passwordMessage");
  let passwordRepeated = document.getElementById("passwordRepeated");
  let passwordRepeatedMessage = document.getElementById("passwordRepeatedMessage");
  passwordMessage.textContent = "";
  passwordRepeatedMessage.textContent = "";
  if (accountForm.password.value.length < 8) {
    passwordMessage.textContent = "Le mot de passe doit contenir au moins 8
caractères.";
  } else if (accountForm.password.value != passwordRepeated.value) {
    passwordRepeatedMessage.textContent = "Les mots de passe ne correspondent pas.";
  } else {
    let rawValue = accountForm.password.value;
    accountForm.password.value = await hashSHA256(rawValue);
    accountForm.submit();
    accountForm.password.value = rawValue;
  }
}
```

Les scripts de account.ejs

```
async function validateAccountCreation() {
  let passwordMessage = document.getElementById("passwordMessage");
  let passwordRe                etElementById("passwordRepeated");
  let passwordRepeatedMessage = document.getEleme
  passwordMessage.textContent = "";
  passwordRepeatedMessage.textContent = "";
  if (accountForm.password.value.length < 8) {
    passwordMessage.textContent = "Le mot de passe doit contenir au moins 8
caractères.";
  } else if (accountForm.password.value != passwordRepeated.value) {
    passwordRepeatedMessage.textContent = "Les mots de passe ne correspondent pas.";
  } else {
    let rawValue = accountForm.password.value;
    accountForm.password.value = await hashSHA256(rawValue);
    accountForm.submit();
    accountForm.password.value = rawValue;
  }
}
```

Vérifications classiques

Dans la requête, on ne passe pas le mot de passe en clair. Seule son empreinte est transmise.
Cela protège les personnes qui utilisent le même mot de passe sur plusieurs sites.

On cache l'empreinte.

Le script hash.js

```
async function hashSHA256(input) {
  let textBuffer = new TextEncoder().encode(input);
  let hashBuffer = await window.crypto.subtle.digest("SHA-256", textBuffer);
  const hashArray = Array.from(new Uint8Array(hashBuffer));
  const hash = hashArray
    .map((item) => item.toString(16).padStart(2, "0"))
    .join("");
  return hash;
}
```

Réponse du serveur

- Si le nom d'utilisateur n'est pas disponible, on demande à l'utilisateur de recommencer.

```
export function createAccount(req, res) {
  let { username, password } = req.body;
  let user = forum.users.find((user) => user.username == username);
  if (user) {
    res.render("account", {
      username,
      message: "Ce nom n'est pas disponible.",
    });
  } else {
    // ...
}
```

Si la demande avait été faite directement en utilisant l'API, l'entête http `Accept: application/json` aurait été défini.
En ce cas, la réponse attendue serait plutôt `res.json({ success: false });`

Réponse du serveur

- Sinon, on enregistre l'utilisateur.

```
export function createAccount(req, res) {
  let { username, password } = req.body;
  let user = forum.users.find((user) => user.username == username);
  if (user) { /* ... */ } else {
    let user = {
      id: getNewId(),
      username,
      password: createHash("sha256").update(password).digest("hex"),
    };
    forum.users.push(user);
    // ...
    res.redirect("/");
  }
}
```

Réponse du serveur

- Sinon, on enregistre l'utilisateur.

```
export function createAccount(req, res) {
  let { username, password } = req.body;
  let user = forum.users.find((use
  if (user) { /* ... */ } else {
    let user = {
      id: getNewId(),
      username,
      password: createHash("sha256").update(password).digest("hex"),
    };
    forum.users.push(user);
    // ...
    res.redirect("/");
  }
}
```

On sauvegarde l'empreinte de l'empreinte du mot de passe.
Cela empêche la connexion si la base de données est compromise.
La fonction `createHash` vient du module standard `"node:crypto"`.

On est ensuite redirigé vers la page d'accueil du forum.

Authentification

- Il existe plusieurs façons de maintenir l'authentification d'un utilisateur.
- On présente ici une méthode basée sur les JSON Web Tokens (JWT).
- Il s'agit de données JSON cryptées et transmises par le serveur au client.
- Le client renvoie ce jeton à chaque requête.
- Le serveur peut vérifier l'identité du client en décryptant le jeton.
- Les jetons ont une durée de vie limitée pour des questions de sécurité.
- On utilise l'implémentation issue du package `jsonwebtoken`.

```
import jwt from "jsonwebtoken";
```

Authentification

- La première action à réaliser est de choisir une clé secrète.
- On peut générer automatiquement une clé secrète dans node à l'aide de la commande :

```
require('crypto').randomBytes(64).toString('hex')
```

- On stocke usuellement la clé obtenue dans un fichier de configuration (fichier .env à la racine du serveur).

```
# .env configuration file
SECRET=823ad7dd32962a8cada3be30ffdb4dfaf69ed75828646cda945756190643bfd5e7dfb8369496f2a
54e993a7ca548388303b3734b06c338aebee5f591debda6bd
```

Authentification

- Pour récupérer la clé secrète, on peut utiliser le module `dotenv`. Celui-ci va analyser le fichier `.env` et ajouter les variables qui s'y trouvent dans l'objet `process.env`.

```
import dotenv from "dotenv";
dotenv.config(); // parsing du fichier .env
```

- On peut désormais créer des JWT :

```
function createJWT(user) {
  return jwt.sign(
    { id: user.id, username: user.username }, // données à crypter
    process.env.SECRET, // clé de chiffrement
    { expiresIn: "7d" }, // durée de validité du jeton, ici une semaine
  );
}
```

Authentification

- Il faut transmettre ce jeton à l'utilisateur qui devra le renvoyer à chaque requête au serveur.
- Option n°1 :
  - Le serveur l'envoie au format JSON via `res.json(token)`.
  - Dans ses requêtes, le client ajoute l'entête `Authorization: Bearer MY_TOKEN`.
- Option n°2 :
  - Le serveur l'envoie dans un cookie via `res.cookie("cookieName", token)`.
  - Dans ses requêtes, le client renvoie le cookie. Cette opération est réalisée automatiquement par les navigateurs.
- On privilégie ici la seconde option, plus simple à mettre en œuvre.

Authentification

- Pour s'authentifier automatiquement à la création du compte, il suffit donc d'ajouter ses lignes :

```
export function createAccount(req, res) {
  let { username, password } = req.body;
  let user = forum.users.find((user) => user.username == username);
  if (user) { /* ... */ } else {
    let user = /* ... */;
    forum.users.push(user);
    let token = createJWT(user);
    res.cookie("accessToken", token, { httpOnly: true });
    res.redirect("/");
  }
}
```

Rend les cookies invisibles depuis les scripts JS.
Augmente la sécurité pour contrer certaines attaques.

Authentification

- Le serveur doit analyser les cookies entrant pour vérifier l'identité de l'utilisateur.
- On utilise le middleware `cookie-parser` pour analyser les cookie.
- On installe ensuite notre propre middleware d'authentification.

```
import cookieParser from "cookie-parser";
app
  // ...
  .use(cookieParser()) // peuple req.cookies
  .use(routes.authenticate) // initialise res.locals.user
  // Les routes nécessitant une authentification viennent après.
```

Authentification

- La fonction `authenticate` se résume à vérifier la présence et la validité du jeton d'accès pour initialiser `res.locals.user`.

```
export function authenticate(req, res, next) {
  try {
    let token = req.cookies.accessToken;
    let user = jwt.verify(token, process.env.SECRET);
    res.locals.user = user;
  } catch {}
  next();
}
```

- À partir de maintenant, si `res.locals.user` est défini, alors l'utilisateur est authentifié.

Authentification

- On peut modifier les pages renvoyées à l'utilisateur en fonction de son état authentifié / non authentifié.

```
<!-- Entête du forum (inclus dans index.ejs et thread.ejs) -->
<% if (locals.user) { %>
<p>Bienvenue utilisateur : <%= locals.user.username %></p>
<div><a href="/account/logout" class="button">Se déconnecter</a></div>
<% } else { %>
<table>
  <tbody>
    <tr><td><a href="/account/login" class="button">Se connecter</a></td>
        <td><a href="/account/new" class="button">Nouveau compte</a></td></tr>
  </tbody>
</table>
<% } %>
```

Authentification

- La connexion suit le même modèle que la création de compte.

```
export function login(req, res) {
  let { username, password } = req.body;
  let user = forum.users.find((user) => user.username == username);
  if (user && user.password == createHash("sha256").update(password).digest("hex")) {
    let token = createJWT(user);
    res.cookie("accessToken", token, { httpOnly: true });
    res.redirect("/");
  } else {
    res.render("login", { message: "Nom d'utilisateur/mot de passe invalide." });
  }
}
```

Authentification

- Pour la déconnexion, il suffit de supprimer le cookie.

```
export function logout(req, res) {
  res.cookie("accessToken", null);
  res.redirect("/");
}
```

- Attention, cette méthode est simpliste. Si le jeton a été intercepté, il reste valide jusqu'à son expiration.

Compléter le serveur

- Il nous reste à compléter le serveur.
- Il faudrait à minima ajouter les routes suivantes :

```
app
  // ...
  .get("/threads/new", routes.getThreadCreationPage)
  .post("/threads/new", routes.createThread)
  .post("/threads/:threadId/", routes.replyThread)
  .get("/threads/:threadId/:postId", routes.getPostModificationPage)
  .put("/threads/:threadId/:postId", routes.modifyPost)
  .delete("/threads/:threadId/:postId", routes.deletePost)
```

PUT et DELETE

- Comment générer les requêtes PUT et DELETE ?
- Solution n°1 :
  - Créer un élément et écouter un évènement particulier.
  - Dans l'écouteur, utiliser la fonction `fetch` en précisant la méthode désirée.
- Inconvénients :
  - On doit gérer explicitement les entêtes et le corps de la requête.
  - Si on utilise un formulaire pour récolter des données, on court-circuite le fonctionnement normale.

PUT et DELETE

- Comment générer les requêtes PUT et DELETE ?
- Solution n°2 :
  - Utiliser une requête POST.
  - Préciser la méthode réellement souhaitée dans la chaîne de requête ou le corps.
  - Le serveur analyse la requête pour rétablir la bonne méthode.
- Avantages :
  - Compatible avec les formulaires.
  - Procédé connu et implémenté automatiquement par certains packages.

PUT et DELETE : exemple

```
<form
  name="modifyPost"
  action="/threads/<%= params.threadId %>/<%= params.postId %>?_method=PUT"
  method="POST">
  <table>
    <tbody>
      <tr><td><label for="content">Message</label></td>
          <td><textarea name="content" rows="10" cols="100">
                <%= post.content %>
              </textarea></td></tr>
      <tr><td><input type="submit" value="Modifier" /></td></tr>
    </tbody>
  </table>
</form>
```

Lorsque le formulaire est soumis, la chaîne de requête indique la méthode à utiliser (ici associée au paramètre _method).

PUT et DELETE : exemple

```
import methodOverride from "method-override";


app
  // ...
  .use(methodOverride("_method"))
  // À partir d'ici la méthode a été modifiée si _method était défini dans la chaîne
de requête.
```

- Fonctionne pour les chaînes de requête et les entêtes.
- Peut s'adapter pour faire des tests plus complexes :

```
app.use(methodOverride((req, res) => {
    // ...
    return "DELETE";
  }));
```

Et maintenant ?

- Rationnaliser l'organisation du code.
- Ajouter la persistance avec une base de données (par exemple MongoDB interfacé avec le package `mongoose`).
- Passer à un framework front-end plus avancé (Angular, React, Vue, …).
- Améliorer le cycle de développement :
  - utiliser `nodemon` pour un redémarrage automatique du serveur,
  - ajouter des scripts pour changer de configuration,
  - …