

Programmation Web côté serveur

DÉVELOPPEMENT WEB PLUS FACILE AVEC EXPRESS

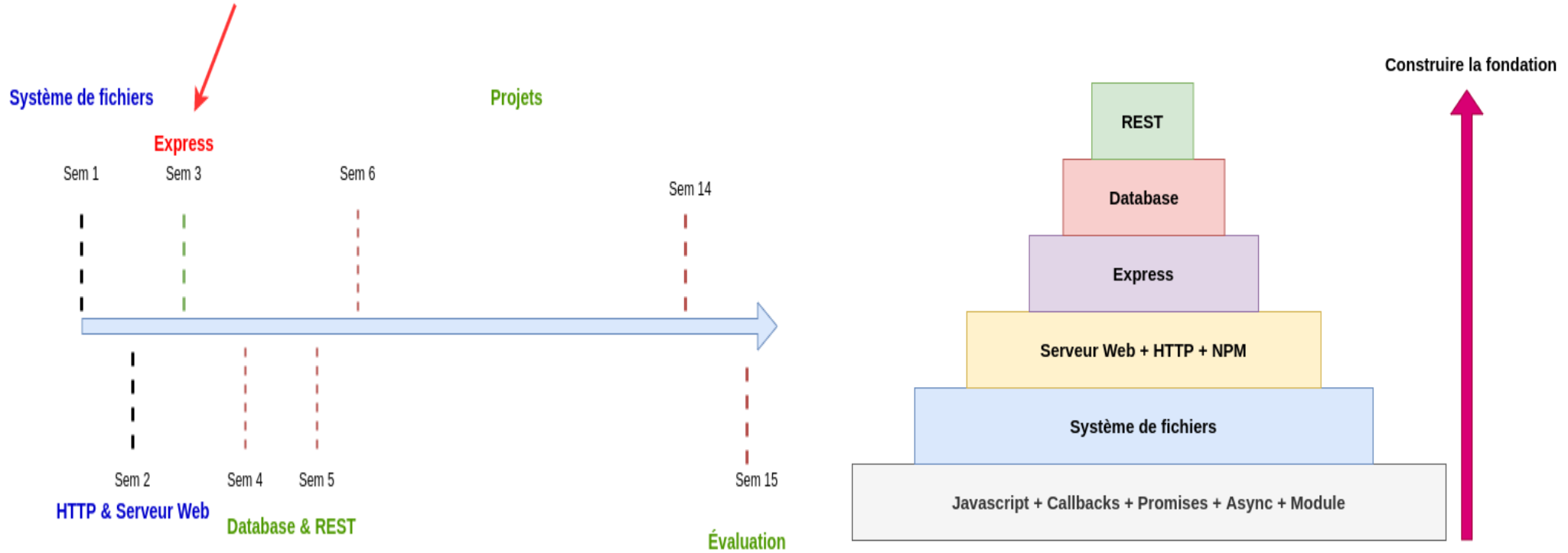
20-sept-2021

Joseph AZAR

Cette leçon couvre

- Présentation d'Express
- Routage avec Express
- Introduction aux concepts de mises en page (layouts) et de vues rendues dynamiquement

Avancement du cours



Qu'est-ce qu'un framework web

Un framework Web dans Node.js est un module qui offre une structure à votre application. Grâce à cette structure, vous pouvez facilement créer et personnaliser le fonctionnement de votre application sans vous soucier de créer certaines fonctionnalités à partir de zéro, telles que le service de fichiers individuels.

Ce que nous avons fait dans le dernier TP est bon mais pas optimal.
Nous pourrions le faire plus facilement avec un framework web

```
utils.js x contentTypes.js x main.js x router.js x
1  const port = 3000,
2      http = require("http"),
3      httpStatus = require("http-status-codes"),
4      router = require("./router"),
5      contentTypes = require("./contentTypes"),
6      utils = require("./utils");
7
8  router.get( url: "/", action: (req, res) => {
9      res.writeHead(httpStatus.OK, contentTypes.html);
10     utils.getFile( file: "views/index.html", res);
11 });
12
13 router.get( url: "/courses.html", action: (req, res) => {
14     res.writeHead(httpStatus.OK, contentTypes.html);
15     utils.getFile( file: "views/courses.html", res);
16 });
17
18 router.get( url: "/contact.html", action: (req, res) => {
19     res.writeHead(httpStatus.OK, contentTypes.html);
20     utils.getFile( file: "views/contact.html", res);
21 });
22
23 router.get( url: "/courses", action: (req, res) => {
24     res.writeHead(httpStatus.OK, contentTypes.html);
25     utils.getFile( file: "views/courses.html", res);
26 });
27
28 router.get( url: "/contact", action: (req, res) => {
29     res.writeHead(httpStatus.OK, contentTypes.html);
30     utils.getFile( file: "views/contact.html", res);
31 });
32
33
```

Les frameworks Node.js à connaître



Express.js augmente la vitesse de développement et fournit une structure stable sur laquelle créer des applications



Construit sur Express.js, offrant plus de structure, ainsi qu'une bibliothèque plus grande et moins de possibilités de personnalisation (<https://sailsjs.com/>)



Conçu par des développeurs qui ont créé Express.js en mettant l'accent sur une bibliothèque de méthodes non proposées dans Express.js (<http://koa.js.com/>)



Conçu avec une architecture similaire à Express.js et axé sur l'écriture de moins de code (<https://hapijs.com/>)



Axé sur la fourniture de la meilleure expérience de développement avec une puissante architecture de plug-in (<https://www.fastify.io/>)



Construit sur le module HTTP de base et acclamé pour sa gestion des requêtes et des réponses haute performance (<https://www.totaljs.com/>)

Express

Express.js est le framework le plus utilisé dans la communauté Node.js, vous assurant de trouver le support dont vous avez besoin par rapport au support offert par d'autres frameworks plus récents

Express.js fournit des méthodes et des modules pour aider à gérer les demandes, à servir du contenu statique et dynamique, à connecter des bases de données et à suivre l'activité des utilisateurs

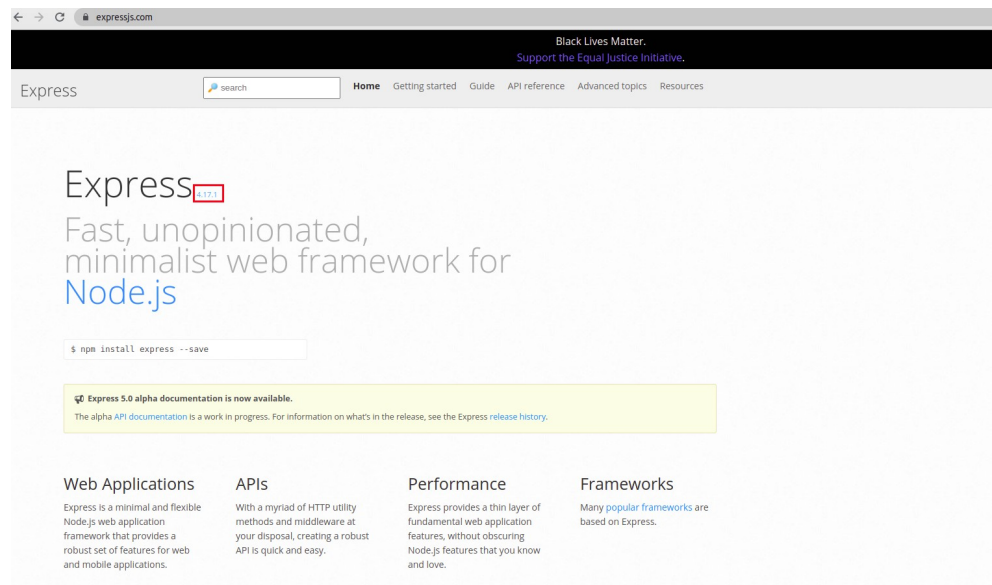
Étant donné qu'Express.js est un package externe, il n'est pas préinstallé avec Node.js. Vous devez le télécharger et l'installer en exécutant la commande suivante dans le répertoire de votre projet dans le terminal : **npm install express --save**

Express

Au moment d'écrire ces lignes, la dernière version d'Express.js est la 4.17.1. Pour vous assurer que votre version d'Express.js est cohérente avec celle utilisée dans cette présentation, installez le package en exécutant

`npm install express@4.17.1 --save .`

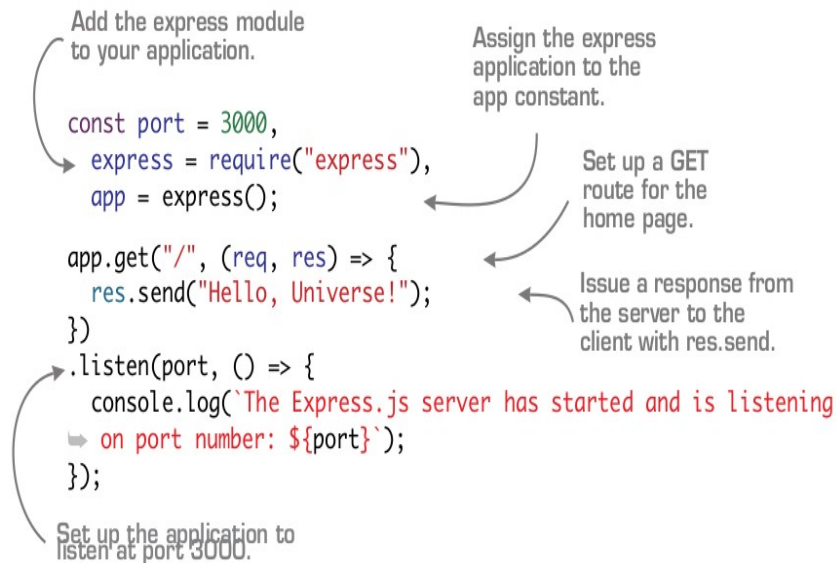
Utilisez l'indicateur **--save/-S** pour qu'Express.js soit répertorié en tant que dépendance d'application. En d'autres termes, votre application dépend d'Express.js pour fonctionner, vous devez donc vous assurer qu'elle est installée.



Construire votre première application Express.js

Commencer par **const express = require('express');** pour importer le module express. Ensuite, écrivez **const app = express();**. Nous pouvons maintenant utiliser les méthodes de la variable d'application pour définir les aspects de notre service Web.

- **app.get:** nous permet de créer un point de terminaison GET. Il prend deux arguments: le chemin de l'URL du point de terminaison et une fonction qui définit le comportement
- **req:** est l'objet de la requête et contient des éléments tels que les paramètres de la requête
- **res:** est l'objet de réponse et a des méthodes pour envoyer des données au client
- **res.set(...):** définit les données d'en-tête, comme **"content-type"**.
- **res.send(response):** renvoie la réponse sous forme de texte au client.
- **res.json(response):** fait de même, mais avec un objet JSON.



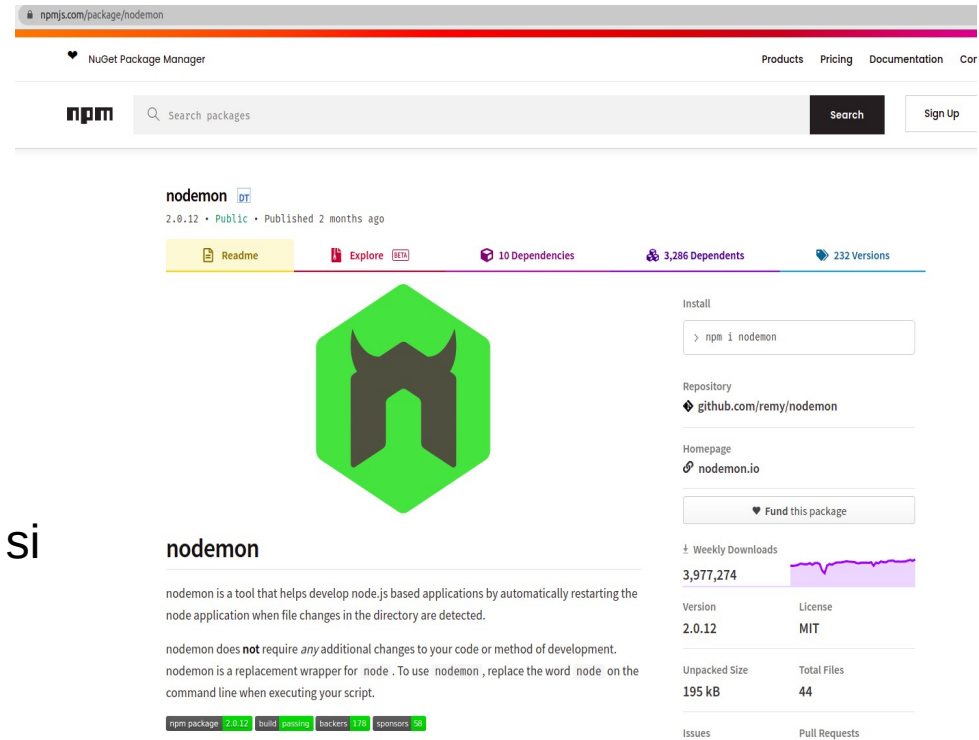
```
app.get('/hello', function (req, res) {  
  res.set("Content-Type", "text/plain");  
  res.send('Hello World!');  
});
```


Lancer le serveur avec nodemon

Vous pouvez installer nodemon en tant que dépendance de développement (devDependency) ou en tant que ressource que vous utilisez uniquement lors du développement d'une application:

npm i nodemon --save-dev

nodemon est une version de node qui redémarre si vous apportez des modifications au code JS pour refléter les modifications

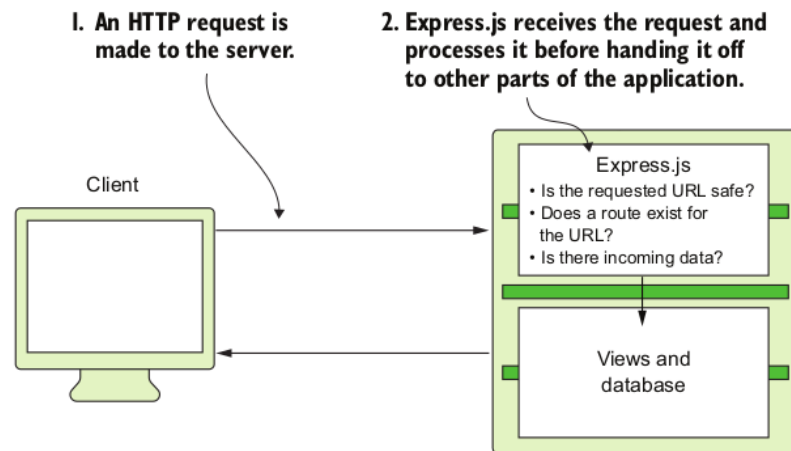


Middlewares

Un framework Web comme Express.js fonctionne via des fonctions considérées comme des middleware car elles se situent entre l'interaction HTTP sur le Web et la plate-forme Node.js

Middleware est un terme général appliqué au code qui aide à écouter, analyser, filtrer et gérer la communication HTTP avant que les données n'interagissent avec la logique de l'application

Vous pouvez considérer le middleware comme un bureau de poste. Avant que votre colis puisse entrer dans le réseau de livraison, un employé des postes doit inspecter la taille de votre boîte et s'assurer qu'elle est correctement payée et qu'elle respecte les politiques de livraison



Middlewares

En plus de la méthode “send” sur l'objet de réponse, Express.js fournit des moyens plus simples d'extraire les données du corps de la requête

- **params**: vous permet d'extraire les identifiants et les tokens de l'URL
- **body**: contient une grande partie du contenu de la requête, qui comprend souvent des données provenant d'une requête POST, comme un formulaire soumis. A partir du corps de la requête, vous pouvez collecter des informations rapidement et les enregistrer dans une base de données
- **URL**: fournit des informations sur l'URL visitée
- **query** : comme body, vous permet d'extraire les données soumises au serveur d'applications. Il est souvent demandé dans l'URL en tant que chaîne de requête (query string)

```
const port = 3000,  
      express = require("express"),  
      app = express();  
  
app.get("/", (req : Request<P, ResBody, ReqBody, ReqQuery, Locals> , res : Response<ResBody, Locals> ) => {  
  console.log(req.params);  
  console.log(req.body);  
  console.log(req.url);  
  console.log(req.query);  
  res.send( body: "Bonjour, $3!");  
})
```

```
"use strict";  
// 1. Charger les modules requis  
const express = require("express");  
const app = express();  
  
// 2. Ajoutez des routes et d'autres middleware et fonctions ici  
  
// 3. Démarrez l'application sur un port ouvert!  
const PORT = 3000;  
app.listen(PORT);
```

Routage en Express

Vous pouvez utiliser les méthodes HTTP sur l'objet **app** car **app** est une instance de la classe de framework Express.js principale. En installant ce package, vous avez hérité des méthodes de routage sans avoir besoin d'écrire d'autre code

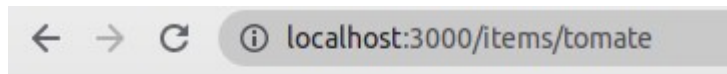
```
app.post("/contact", (req, res) => {  
  res.send("Contact information submitted successfully.");  
});
```

Handle requests with the Express.js post method.

```
app.get("/items/:vegetable", (req, res) => {  
  res.send(req.params.vegetable);  
});
```

Respond with path parameters.

```
app.get("/items/:vegetable", (req : Request<P, ResBody, ReqBody, ReqQuery, Locals> , res : Response<ResBody, Locals> ) => {  
  res.set("content-types", "text/html")  
  res.send( body: `<h1>${req.params.vegetable}</h1>` );  
});
```



tomate

Path Parameters

Les paramètres de chemin permettent à un utilisateur de transmettre des "variables" à un point de terminaison (endpoint)

Vous pouvez définir un paramètre de route avec: **param**

Route path: `/states/:state/cities/:city`

Request URL: `http://localhost:8000/states/wa/cities/Seattle`

req.params: `{ "state": "wa", "city": "Seattle" }`

```
app.get("/states/:state/cities/:city", function (req : Request<P, ResBody, ReqBody, ReqQuery, Locals> ,  
res : Response<ResBody, Locals> ) {  
  
  let state = req.params.state; // wa  
  let city = req.params.city;   // Seattle  
  res.set("content-types", "text/html")  
  res.send( body: `<h1>${state} - ${city}</h1>` );  
});
```

localhost:3000/states/Franche-Comté/cities/Belfort

Franche-Comté - Belfort

Query Parameters

Vous pouvez également utiliser des paramètres de requête (query string) dans Express à l'aide de l'objet **req.query**, bien qu'ils soient plus utiles pour les paramètres facultatifs

Route path: /cityInfo

Request URL: http://localhost:8000/cityInfo?state=wa&city=Seattle

req.query: { "state": "wa", "city": "Seattle" }

```
app.get("/cityInfo", function (req : Request<P, ResBody, ReqBody, ReqQuery, Locals> ,  
                                res : Response<ResBody, Locals> ) {  
  
    let state = req.query.state;  
    let city = req.query.city;  
    res.set("content-types", "text/html")  
    res.send( body: `<h1>${state} - ${city}</h1>` );  
});
```

← → ↻ ⓘ localhost:3000/cityInfo?state=Franche-Comté&city=Belfort

Franche-Comté - Belfort

Chaîne dans le cycle requête-réponse

next est fourni comme un moyen d'appeler la fonction suivante dans votre flux d'exécution de requête-réponse. A partir du moment où une requête entre dans le serveur, elle accède à une série de fonctions middleware. Selon l'endroit où vous ajoutez votre propre fonction middleware personnalisée, vous pouvez utiliser **next** pour informer Express.js que votre fonction est terminée et que vous souhaitez continuer avec la fonction suivante dans la chaîne.

```
app.use((req, res, next) => {  
  console.log(`request made to: ${req.url}`);  
  next();  
});
```

Define a middleware function.

Log the request's path to console.

Call the next function.

L'appel de **next** à la fin de votre fonction est nécessaire pour alerter Express.js que votre code est terminé. Ne pas le faire laisse votre demande en suspens. Le middleware s'exécute de manière séquentielle, donc en n'appelant pas **next**, vous empêchez votre code de continuer jusqu'à la fin

<http://expressjs.com/en/guide/using-middleware.html>

Analyser les données de la requête

Vous avez deux manières principales d'obtenir des données de l'utilisateur:

- Via le corps de la requête dans une requête POST
- Via la chaîne de requête (query string) de la requête dans l'URL

POST

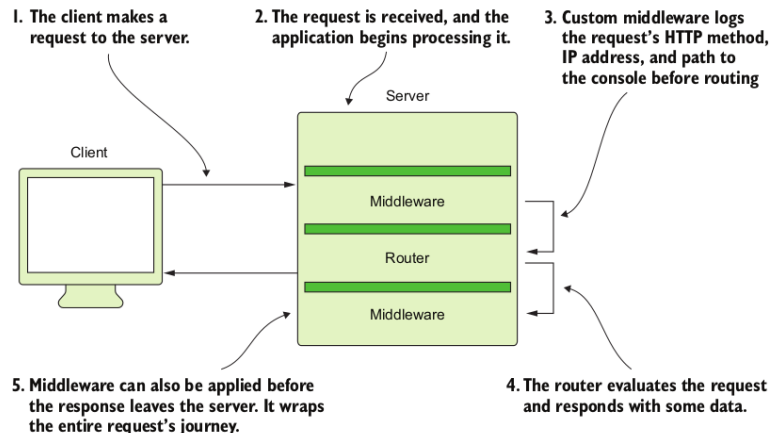
```
app.use(
  express.urlencoded({
    extended: false
  })
);
app.use(express.json());

app.post("/", (req, res) => {
  console.log(req.body);
  console.log(req.query);
  res.send("POST Successful!");
});
```

Tell your Express.js application to parse URL-encoded data.

Create a new post route for the home page.

Log the request's body.



Express.js facilite la récupération du corps de la requête avec l'attribut **body**. Pour faciliter la lecture du contenu du corps, vous ajoutez **express.json** et **express.urlencoded** à votre instance d'application pour analyser les corps de requête entrants

Analyser les données de la requête

```
const port = 3000,
      express = require("express"),
      app = express();

app.use(
  express.urlencoded({
    extended: false
  })
);
app.use(express.json());
app.post(path: "/", handlers: (req : Request<P, ResBody, ReqBody, ReqQuery, Locals> , res : Response<ResBody, Locals> ) => {
  console.log(req.body);
  console.log(req.query);
  res.send( body: "POST Successful! \n");
});

app.listen(port, hostname: () => {
  console.log(`The Express.js server has started and is listening on port number: ${port}`);
});
```

`curl --data "nom=<X>&prenom=<Y>" http://localhost:<port>`

```
(base) $ curl --data "nom=Gates&prenom=Bill" http://localhost:3000
POST Successful!
```

Utilisation de MVC

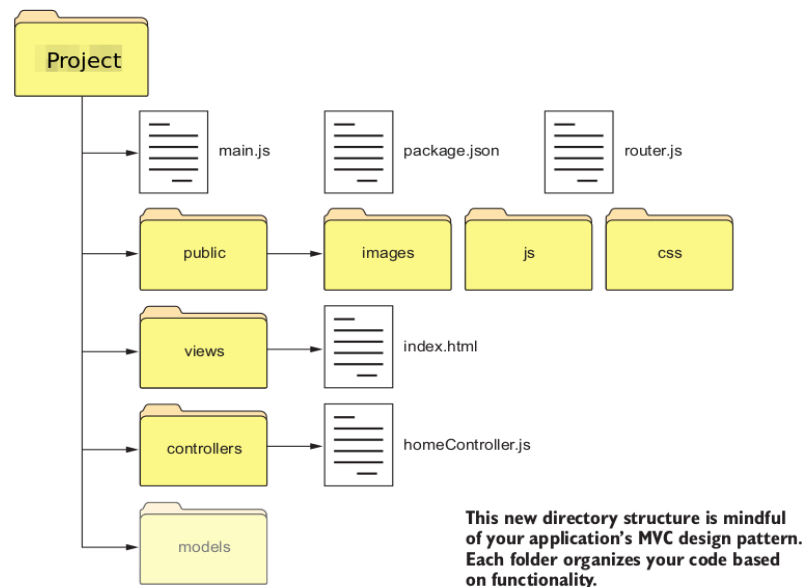
Pour organiser une base de code croissante, vous allez suivre une architecture d'application connue sous le nom de MVC

L'architecture MVC se concentre sur trois parties principales des fonctionnalités de votre application: les modèles, les vues et les contrôleurs

Vues: affichages rendus des données de votre application

Modèles: classes qui représentent des données orientées objet dans votre application et votre base de données. Par exemple, vous pouvez créer un modèle pour représenter une commande client. Dans ce modèle, vous définissez les données qu'une commande doit contenir et les types de fonctions que vous pouvez exécuter sur ces données

Contrôleurs: le lien entre les vues et les modèles. Les contrôleurs exécutent la majeure partie de la logique lorsqu'une requête est reçue pour déterminer comment les données du corps de la requête doivent être traitées et comment impliquer les modèles et les vues. Ce processus devrait sembler familier, car dans une application Express.js, vos fonctions de rappel de route agissent comme des contrôleurs



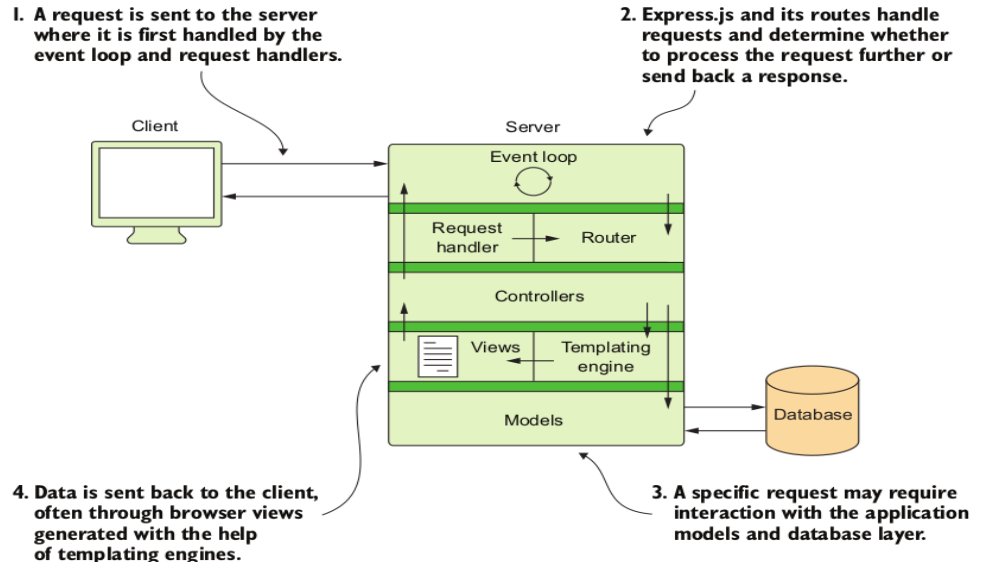
Utilisation de MVC

Pour suivre le modèle de conception MVC, déplacez vos fonctions de rappel (Callbacks) vers des modules distincts qui reflètent les objectifs de ces fonctions. Les fonctions de rappel liées à la création, à la suppression ou aux modifications de compte utilisateur, par exemple, vont dans un fichier appelé **usersController.js** dans le dossier des contrôleurs. Les fonctions pour les routes qui rendent la page d'accueil ou d'autres pages d'informations peuvent aller dans **homeController.js** par convention

homeController.js

```
exports.sendReqParam = (req, res) => {  
  let veg = req.params.vegetable;  
  res.send(`This is the page for ${veg}`);  
};
```

Create a function to handle route-specific requests.



main.js

```
app.get("/items/:vegetable", homeController.sendReqParam);
```

Handle GET requests to "/items/:vegetable".

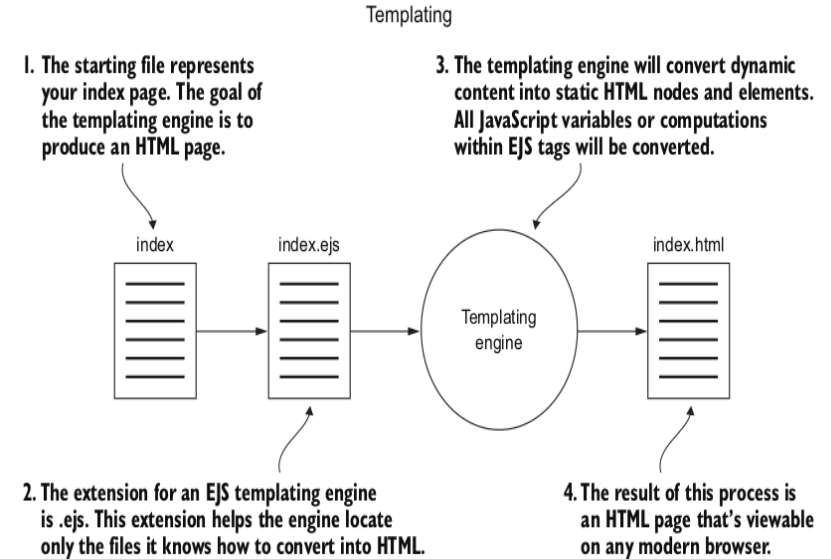
Connexion des vues avec des templates

Considérez ceci: Vous avez des cadres définissant l'apparence de vos pages d'application, et vous remarquez que de nombreuses pages partagent les mêmes composants. Votre page d'accueil et votre page de contact utilisent toutes deux la même barre de navigation. Au lieu de réécrire le code HTML représentant la barre de navigation pour chaque vue, vous souhaitez écrire le code une fois et le réutiliser pour chaque vue.

Avec la création de **templates** dans une application Node.js, c'est exactement ce que vous pouvez faire. En fait, vous pourrez restituer une seule mise en page pour toutes vos pages d'application ou partager le contenu de la vue dans des extraits de code appelés “**partials**” .

Templating engine

- Une partie de ce qui rend Express.js si populaire est sa capacité à fonctionner avec d'autres packages et outils
- L'un de ces outils est le moteur de création de “**templates**”. “**Templating**” vous permet de coder vos vues avec la possibilité d'insérer des données dynamiques
- Vous pouvez écrire vos vues en HTML avec **EJS** - des données sous forme d'objets JavaScript intégrés dans la page avec une syntaxe spéciale
- Ces fichiers ont l'extension **.ejs**



npm install ejs --save


Templating engine

Maintenant que le package **ejs** est installé, vous devez informer votre application Express.js que vous prévoyez de l'utiliser pour la création de templates. Pour ce faire, ajoutez **app.set("view engine", "ejs")**. Cette ligne indique à votre application Express.js de définir son moteur de vue comme ejs. Cette ligne est la façon dont votre application sait s'attendre à EJS dans votre dossier de vues (Views) dans votre répertoire de projet principal.

controllers/homeController.js

```
exports.respondWithName = (req, res) => {  
  res.render("index");  
};
```

views/index.ejs

```
 let name = "Bill"; %>  
<h1> Hello, <%= name %> </h1>
```

main.js

```
const express = require("express");  
const app = express();  
const homeController = require("../Controllers/homeController")  
app.set("view engine", "ejs")  
  
app.get("/", homeController.respondWithName);  
  
const PORT = 3000;  
app.listen(PORT);
```

Transmission des données de vos contrôleurs

la meilleure façon d'utiliser les templates est de transmettre les données de vos contrôleurs à vos vues au lieu de définir ces variables directement dans la vue

controllers/homeController.js

```
exports.respondWithName = (req, res) => {  
  let paramsName = req.params.nom;  
  res.render("index", { name: paramsName });  
};
```

views/index.ejs

```
<h1> Hello, <%= name %> </h1>
```

main.js

```
const express = require("express");  
const app = express();  
const homeController = require("../controllers/homeController")  
app.set("view engine", "ejs")  
  
app.get("/nom/:nom", homeController.respondWithName);  
  
const PORT = 3000;  
app.listen(PORT);
```

← → ↻ ⓘ localhost:3000/nom/Joseph

Hello, Joseph

“Partials” et “Layouts”

Considérez les mises en page (layouts) comme le contenu qui ne change pas d'une page à l'autre lorsque vous naviguez sur un site Web. Le bas (pied de page) de la page ou la barre de navigation peut rester le même, par exemple. Au lieu de recréer le code HTML de ces composants, ajoutez-les à un fichier **layout.ejs** que d'autres vues peuvent partager

Installez le package express-ejs-layouts: [npm install express-ejs-layouts](#)

main.js

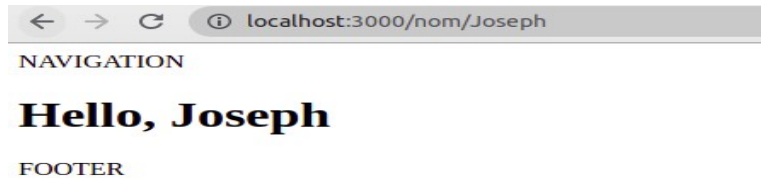
```
const express = require("express");
const app = express();
const homeController = require("../controllers/homeController");
const layouts = require("express-ejs-layouts");
app.set("view engine", "ejs");
app.use(layouts);

app.get("/nom/:nom", homeController.respondWithName);

const PORT = 3000;
app.listen(PORT);
```

views/layout.ejs

```
<body>
<div id="nav">NAVIGATION</div>
<%- body %>
<div id="footer">FOOTER</div>
</body>
```

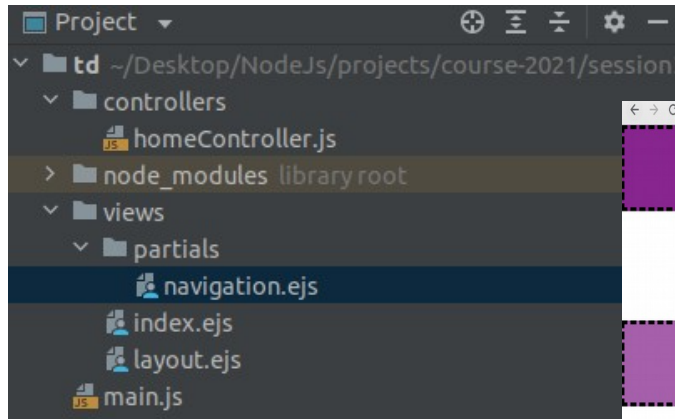


“Partials” et “Layouts”

Les “Partials” fonctionnent de la même manière que les “Layouts”. Les “Partials” sont des extraits de contenu de vue qui peuvent être inclus dans d'autres vues.

Pour créer un “Partial” pour l'élément de navigation, déplacez votre code pour ce div dans un nouveau fichier appelé **navigation.ejs**. Placez ce fichier dans un nouveau dossier appelé **partials** dans votre dossier **views**. Incluez ensuite ce fichier dans votre fichier **layout.ejs** en utilisant le code suivant :`<%- include('partials/navigation.ejs'); %>`

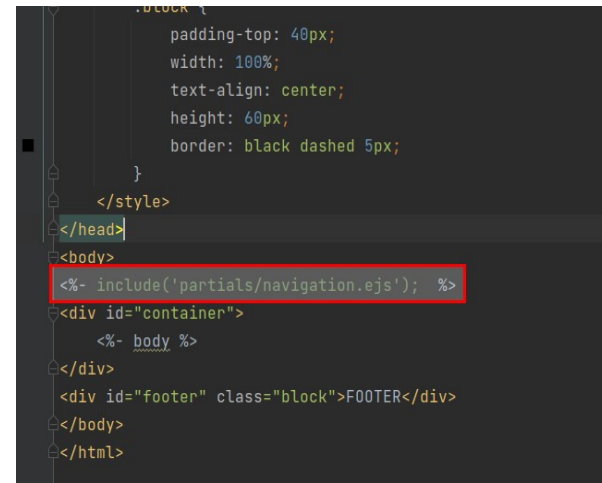
Project



views/partials/navigation.ejs

```
<div id="nav" class="block">TOP NAVIGATION</div>
```

views/layout.ejs



Gestion des erreurs avec Express.js

Vous pouvez adopter quelques approches pour la gestion des erreurs avec Express.js. La première approche consiste à afficher un message sur votre console chaque fois qu'une erreur se produit. Vous pouvez créer le fichier **errorController.js** dans votre dossier de contrôleurs et ajouter des fonctions pour gérer les erreurs

```
const httpStatus = require("http-status-codes");

exports.respondNoResourceFound = (req, res) => {
  let errorCode = httpStatus.NOT_FOUND;
  res.status(errorCode);
  res.send(`${errorCode} | The page does not exist!`);
};

exports.respondInternalServerError = (error, req, res, next) => {
  let errorCode = httpStatus.INTERNAL_SERVER_ERROR;
  console.log(`ERROR occurred: ${error.stack}`)
  res.status(errorCode);
  res.send(`${errorCode} | Sorry, our application is
  experiencing a problem!`);
};
```

Respond with a 404 status code.

Catch all errors and respond with a 500 status code.

Gestion des erreurs avec Express.js

Si vous souhaitez personnaliser vos pages d'erreur, vous pouvez ajouter une page 404.html et une page 500.html dans votre dossier public avec du HTML de base. Ensuite, au lieu de répondre avec un message en texte brut, vous pouvez répondre avec ces fichiers

controllers/errorController.js

```
exports.respondNoResourceFound = (req, res) => {  
  let errorCode = HttpStatus.NOT_FOUND;  
  res.status(errorCode);  
  res.sendFile(`./public/${errorCode}.html`, {  
    root: "./"  
  });  
};
```

Respond with a custom error page.

Send content in 404.html.

main.js

```
const express = require("express");  
const app = express();  
const homeController = require("./controllers/homeController");  
const errorController = require("./controllers/errorController");  
const layouts = require("express-ejs-layouts");  
app.set("view engine", "ejs");  
app.use(layouts);  
  
app.get("/nom/:nom", homeController.respondWithName);  
  
app.use(errorController.respondNoResourceFound);  
app.use(errorController.respondInternalServerError);  
  
const PORT = 3000;  
app.listen(PORT);
```

Servir des fichiers statiques

Si un projet est "full-stack" et contient à la fois des fichiers "statiques" côté client ainsi que vos fichiers Node.js, vous pouvez utiliser le middleware **express.static** pour spécifier le répertoire servant les fichiers statiques

La convention est de mettre vos fichiers statiques dans un répertoire **public**, au même endroit que votre fichier Node.js

Dans votre application du dernier TP, servir tous les différents types de fichiers et d'assets statiques nécessiterait des centaines de lignes de code. Avec Express.js, ces types de fichiers sont pris en compte automatiquement. La seule chose que vous devez faire est de dire à Express.js où trouver ces fichiers statiques

Pour configurer cette tâche, vous devez utiliser la méthode **static** du module express. Cette méthode prend un chemin absolu vers le dossier contenant vos fichiers statiques

```
"use strict";  
const express = require("express");  
const app = express();  
  
app.use(express.static(root: "public"));  
  
const PORT = process.env.PORT || 3000;  
app.listen(PORT);
```