Par l'exemple.

Franck Canonne, formateur développeur web freelance.
https://factotum.bzh

Table des matières

L'auteur :

Franck Canonne (https://factotum.bzh), je suis développeur web depuis le milieu des années 1990. Autodidacte, je fais d'abord de nombreux sites statiques en HTML (le CSS n'était pas encore d'actualité). Par connaissance, je réalise le premier site du festival Solidays (en HTML, ASP (l'ASP de l'époque était une sorte de PHP propriétaire créé par Microsoft, rien à voir avec le framework ASP.NET que nous connaissons aujourd'hui) et Flash) ainsi que du site de l'association « solidarité sida ». Très vite passionné par le Web, je me forme toujours de manière autodidacte à PHP et aux bases de données. C'est le déclic. JavaScript n'était alors utilisé que pour réaliser des effets visuels dans des pages PHP ou HTML, et les développeurs PHP portaient des T-Shirts « je n'aime pas le JavaScript). C'était le début d'une longue carrière.

Ensuite développeur en freelance et dans de diverses web-agency, Je quitte Paris pour me rendre en Bretagne, rejoindre mes origines maternelles. Je délaisse vite les sites Web pour me spécialiser dans les applis et les logiciels Web (plus proche de mes origines scientifiques.

Arrive 2003 avec le CMS (logiciel de gestion de contenu) qui révolutionne depuis le monde du site Web. Je fais alors énormément de sites Web « à la chaine » pour faire bouillir la marmite en utilisant ce CMS qui est devenu très puissant et incontournable, mais le vrai travail de développeur et les lignes de codes me manquent de plus en plus. La passion n'y est plus.

En 2018, je passe un diplôme bac+4 pour valider et faire grandir mes compétences. En parallèle et à la suite de ce diplôme, je me forme à tout ce qui tourne autour de JavaScript (React, Vue.js, Node.js, Express.js…). C'est là que je découvre la puissance de Vue.js et que je deviens un vrai fan.

Rien à voir, mais comme vous pouvez le constater sur mon site Internet, je suis aussi passionné de photographie.

A qui s'adresse cet ouvrage ?

Cet ouvrage, n'est pas vraiment un livre, mais plutôt une liste d'exemple concrets de développement Node.js . Les codes présentés dans cet ouvrage sont libres de droit et vous pouvez les utiliser comme bon vous semble.
Celui-ci s'adresse tant aux étudiants et débutants en Node.js qu'aux développeurs confirmés et chefs de projets Web.

Il va de soi que pour un usage « en production », des notions de HTML et CSS sont requis, ainsi que des bases en persistance des données (BDD) et JavaScript (et Node.js) sont un plus.

Les outils nécessaires à la concrétisation de ces exemples ne sont pas extraordinaires : un ordinateur (idéalement sous Linux, mais j'ai utilisé Windows pour écrire ce livre, et MacOS fonctionne également), un IDE tel que Microsoft VS Code qui est très bon, gratuit et gère l'ensemble des fichiers requis (.js, .html, .css, .jse, .json …) ou JetBrains WebStorm qui est payant mais dispose d'une version « étudiant » gratuite. Ce logiciel est en anglais, mais reste la Rolls des IDE pour JavaScript. Je vous invite d'ailleurs à visiter le site de JetBrains (https://www.jetbrains.com/fr-fr/) pour découvrir la multitude d'outils disponible vous facilitant le travail, et notamment DataGrip (pour gérer tout type de base de données) que j'utilise beaucoup. Sinon, Notepad++ et DevToys pour Windows sont deux logiciels que vous utiliserez beaucoup avec l'expérience…

Les Scripts Node.js

Les sessions en Node.js

La gestion des sessions est cruciale dans le développement web pour maintenir l'état de l'application entre les requêtes de l'utilisateur. Node.js propose plusieurs modules et bibliothèques pour gérer les sessions. L'un des modules les plus populaires pour la gestion des sessions est express-session, qui fonctionne bien avec l'Express.js, un framework web pour Node.js.

Voici comment vous pouvez gérer les sessions avec express-session :

Étape 1: Installer express-session

Avant de commencer, assurez-vous d'installer le module express-session en utilisant npm :

```
npm install express-session
```

Étape 2: Configurer express-session dans votre application

Dans votre fichier principal (par exemple, server.js), configurez express-session comme suit :

```
const express = require('express');
const session = require('express-session');
const app = express();

// Configuration d'express-session
app.use(session({
  secret: 'votre_secret', // Clé secrète pour signer la session ID cookie
  resave: false, // Ne réenregistre pas la session si rien n'a changé
  saveUninitialized: true, // Sauvegarde les sessions même si elles ne sont pas initialisées
  cookie: { secure: false } // Configurations de cookie, ici le mode sécurisé est désactivé (à adapter pour la production)
}));
```

Dans l'exemple ci-dessus, secret est une chaîne utilisée pour signer le cookie de session. Vous devriez utiliser une chaîne de caractères longue et complexe pour garantir la sécurité.

Étape 3: Utilisation des sessions dans vos routes

Maintenant que express-session est configuré, vous pouvez l'utiliser dans vos routes pour stocker et récupérer des données de session.

Par exemple, pour stocker une valeur dans la session :

```
app.get('/login', (req, res) => {
  // Simuler un utilisateur authentifié
  const user = { id: 1, username: 'utilisateur1' };
  // Stocker l'utilisateur dans la session
  req.session.user = user;
  res.send('Utilisateur connecté.');
});
```

Pour récupérer la valeur de la session dans une autre route :

```
app.get('/profile', (req, res) => {
  // Vérifier si l'utilisateur est connecté en vérifiant la session
  if (req.session.user) {
    res.send(`Bienvenue, ${req.session.user.username}!`);
  } else {
    res.send('Vous n\'êtes pas connecté.');
  }
});
```

Dans cet exemple, lorsque l'utilisateur accède à la route /login, l'utilisateur est stocké dans la session. Plus tard, lorsque l'utilisateur accède à la route /profile, l'application vérifie la session pour voir si l'utilisateur est connecté.

N'oubliez pas que cette configuration est basique et vous pouvez personnaliser davantage express-session selon les besoins spécifiques de votre application, comme la configuration des options de cookie, l'utilisation de stockage externe (comme une base de données) pour stocker les sessions, etc.

Authentification en Node.js

L'authentification est un élément essentiel dans le développement d'applications web. Voici un exemple de mise en place d'un système d'authentification basique avec Node.js et Express.js en utilisant un nom d'utilisateur et un mot de passe stockés en mémoire. Dans un environnement de production, vous devriez utiliser une base de données sécurisée pour stocker les informations d'authentification.

Étape 1: Installer les dépendances

```
npm install express express-session body-parser
```

Étape 2: Configurer l'authentification dans votre application

```
const express = require('express');
const session = require('express-session');
const bodyParser = require('body-parser');
const app = express();

// Configuration d'express-session
app.use(session({
  secret: 'votre_secret',
  resave: false,
  saveUninitialized: true
}));

// Utilisation de body-parser pour les données POST
app.use(bodyParser.urlencoded({ extended: true }));

// Stockage temporaire des utilisateurs (en production, utilisez une base de données sécurisée)
const users = [
  { id: 1, username: 'utilisateur1', password: 'motdepasse1' },
  { id: 2, username: 'utilisateur2', password: 'motdepasse2' }
```

```
];

// Middleware pour vérifier si l'utilisateur est connecté
const checkAuthenticated = (req, res, next) => {
  if (req.session.user) {
    next();
  } else {
    res.redirect('/login');
  }
};

// Page de connexion
app.get('/login', (req, res) => {
  res.send(`
    <form method="post" action="/login">
      <input type="text" name="username" placeholder="Nom d'utili-
sateur" required><br>
      <input type="password" name="password" placeholder="Mot de
passe" required><br>
      <button type="submit">Se connecter</button>
    </form>
  `);
});
// Gestion de la soumission du formulaire de connexion
app.post('/login', (req, res) => {
  const { username, password } = req.body;
  const user = users.find(u => u.username === username &&
u.password === password);
  if (user) {
    req.session.user = user;
    res.redirect('/profile');
  } else {
    res.send('Identifiants incorrects. Veuillez réessayer.');
  }
});

// Page du profil (accessible uniquement aux utilisateurs authentifiés)
app.get('/profile', checkAuthenticated, (req, res) => {
```

```
    const user = req.session.user;
    res.send(`Bienvenue, ${user.username}! <a href="/logout">Se déconnecter</a>`);
});
// Déconnexion
app.get('/logout', (req, res) => {
    req.session.destroy(err => {
        if (err) {
            console.error(err);
        } else {
            res.redirect('/login');
        }
    });
});
const PORT = 3000;
app.listen(PORT, () => {
    console.log(`Serveur en cours d'exécution sur le port ${PORT}`);
});
```

Dans cet exemple, nous utilisons une structure de données users pour stocker les noms d'utilisateur et les mots de passe. Lorsque l'utilisateur se connecte avec succès, ses informations sont stockées dans la session et il est redirigé vers la page de profil.
L'accès à la page de profil est protégé par le middleware checkAuthenticated, qui redirige les utilisateurs non authentifiés vers la page de connexion.
La déconnexion est gérée en détruisant la session de l'utilisateur.
N'oubliez pas qu'il s'agit d'un exemple basique à des fins éducatives.
En production, vous devriez utiliser des méthodes d'authentification plus robustes, telles que l'authentification basée sur des tokens (JWT), OAuth, ou des bibliothèques d'authentification tierces comme Passport.js (que nous verrons dans l'exemple suivant).
Assurez-vous également de stocker les informations d'identification de manière sécurisée, par exemple dans une base de données chiffrée.

Pour ajouter MySQL à votre application d'authentification Node.js, vous pouvez utiliser le module mysql pour interagir avec la base de données MySQL.
Assurez-vous d'installer ce module en exécutant npm install mysql.
Voici comment vous pouvez mettre en place la connexion à MySQL dans votre

application d'authentification :

Étape 1: Configurer la connexion MySQL.

```
const mysql = require('mysql');
const connection = mysql.createConnection({
  host: 'votre_hôte_mysql',
  user: 'votre_nom_utilisateur',
  password: 'votre_mot_de_passe',
  database: 'votre_base_de_donnees'
});
connection.connect((err) => {
  if (err) {
    console.error('Erreur de connexion à la base de données : ' + err.
stack);
    return;
  }
  console.log('Connecté à la base de données MySQL');
});
module.exports = connection;
```

Étape 2: Mettre à jour le code d'authentification pour utiliser MySQL.

```
const express = require('express');
const session = require('express-session');
const bodyParser = require('body-parser');
const db = require('./db'); // Importer la connexion à MySQL
const app = express();

// Configuration d'express-session
app.use(session({
  secret: 'votre_secret',
  resave: false,
  saveUninitialized: true
}));
app.use(bodyParser.urlencoded({ extended: true }));

// Page de connexion
```

```
app.get('/login', (req, res) => {
  res.send(`
    <form method="post" action="/login">
      <input type="text" name="username" placeholder="Nom d'utili-
sateur" required><br>
      <input type="password" name="password" placeholder="Mot de
passe" required><br>
      <button type="submit">Se connecter</button>
    </form>
  `);
});

// Gestion de la soumission du formulaire de connexion
app.post('/login', (req, res) => {
  const { username, password } = req.body;
  const query = 'SELECT * FROM users WHERE username = ? AND
password = ?';
  db.query(query, [username, password], (err, result) => {
    if (err) {
      console.error(err);
      res.send('Erreur de base de données');
      return;
    }
    if (result.length > 0) {
      const user = result[0];
      req.session.user = user;
      res.redirect('/profile');
    } else {
      res.send('Identifiants incorrects. Veuillez réessayer.');
    }
  });
});

// Page du profil (accessible uniquement aux utilisateurs authentifiés)
app.get('/profile', (req, res) => {
  if (req.session.user) {
    const user = req.session.user;
    res.send(`Bienvenue, ${user.username}! <a href="/logout">Se décon-
```

```
necter</a>`);
    } else {
        res.redirect('/login');
    }
});

// Déconnexion
app.get('/logout', (req, res) => {
    req.session.destroy((err) => {
        if (err) {
            console.error(err);
        } else {
            res.redirect('/login');
        }
    });
});

const PORT = 3000;
app.listen(PORT, () => {
    console.log(`Serveur en cours d'exécution sur le port ${PORT}`);
});
```

Dans cet exemple, nous avons mis à jour le code d'authentification pour exécuter une requête SQL pour vérifier les informations d'authentification dans la base de données MySQL. Assurez-vous d'ajuster le schéma de votre base de données (users dans cet exemple) selon vos besoins.

N'oubliez pas de sécuriser vos requêtes SQL pour prévenir les injections SQL. Vous pourriez utiliser des requêtes préparées ou des requêtes paramétrées pour cela.

Authentification Node.js avec Passport.js et mySQL

Voici un exemple de mise en place de l'authentification avec Passport.js utilisant MySQL comme base de données.

Étape 1: Installez les modules nécessaires

```
npm install express express-session passport passport-local mysql body-parser
```

Étape 2: Configurez la connexion à la base de données MySQL

```
const mysql = require('mysql');
const connection = mysql.createConnection({
  host: 'votre_hôte_mysql',
  user: 'votre_nom_utilisateur',
  password: 'votre_mot_de_passe',
  database: 'votre_base_de_donnees'
});

connection.connect((err) => {
  if (err) {
    console.error('Erreur de connexion à la base de données : ' + err.stack);
    return;
  }
  console.log('Connecté à la base de données MySQL');
});
module.exports = connection;
```

Étape 3: Configurer Passport.js avec la stratégie locale et MySQL

```
const express = require('express');
const passport = require('passport');
const LocalStrategy = require('passport-local').Strategy;
const session = require('express-session');
const bodyParser = require('body-parser');
const db = require('./db'); // Importez votre connexion à MySQL
const app = express();

app.use(session({
  secret: 'votre_secret',
  resave: true,
  saveUninitialized: true
}));

app.use(passport.initialize());
app.use(passport.session());
app.use(bodyParser.urlencoded({ extended: true }));

passport.use(new LocalStrategy(
  (username, password, done) => {
    db.query('SELECT * FROM users WHERE username = ?', [user-
name], (err, results) => {
      if (err) throw err;
      if (results.length === 0) {
        return done(null, false, { message: 'Nom d\'utilisateur incorrect.'
});
      }

      const user = results[0];
      if (password === user.password) {
        return done(null, user);
      } else {
        return done(null, false, { message: 'Mot de passe incorrect.' });
      }
    });
  }
));
```

```
passport.serializeUser((user, done) => {
  done(null, user.id);
});

passport.deserializeUser((id, done) => {
  db.query('SELECT * FROM users WHERE id = ?', [id], (err, results) =>
{
    if (err) throw err;
    const user = results[0];
    done(null, user);
  });
});

app.get('/login', (req, res) => {
  res.send(`
    <form method="post" action="/login">
      <input type="text" name="username" placeholder="Nom d'utili-
sateur" required><br>
      <input type="password" name="password" placeholder="Mot de
passe" required><br>
      <button type="submit">Se connecter</button>
    </form>
  `);
});

app.post('/login', passport.authenticate('local', {
  successRedirect: '/profile',
  failureRedirect: '/login',
  failureFlash: true
}));

app.get('/profile', (req, res) => {
  if (req.isAuthenticated()) {
    res.send(`Bienvenue, ${req.user.username}! <a href="/logout">Se
déconnecter</a>`);
  } else {
    res.redirect('/login');
  }
```

```
});

app.get('/logout', (req, res) => {
  req.logout();
  res.redirect('/login');
});

const PORT = 3000;
app.listen(PORT, () => {
  console.log(`Serveur en cours d'exécution sur le port ${PORT}`);
});
```

Dans cet exemple, nous utilisons Passport.js avec la stratégie d'authentification locale. Lorsque l'utilisateur soumet ses informations de connexion, Passport.js vérifie ces informations dans la base de données MySQL et authentifie l'utilisateur en conséquence. Assurez-vous d'ajuster les requêtes SQL et le modèle de données (users) en fonction de votre base de données.

Créer un CRUD (Create, Read, Update, Delete) en utilisant Node.js et MySQL :

Étape 1 : Installer les dépendances

Assurez-vous d'avoir Node.js installé sur votre système. Ensuite, créez un dossier pour votre projet, accédez à ce dossier dans votre terminal et exécutez les commandes suivantes pour installer les dépendances nécessaires :

```
npm install express mysql body-parser
```

Étape 2 : Configuration de la base de données

Créez un fichier db.js pour configurer la connexion à la base de données MySQL :

```
const mysql = require('mysql');
const connection = mysql.createConnection({
  host: 'localhost',
  user: 'votre_nom_utilisateur',
  password: 'votre_mot_de_passe',
  database: 'nom_de_votre_base_de_donnees'
});

connection.connect((err) => {
  if (err) {
    console.error('Erreur de connexion à la base de données : ' + err.
stack);
    return;
  }
  console.log('Connecté à la base de données MySQL');
```

```
});
module.exports = connection;
```

Étape 3 : Création de l'API CRUD

Créez un fichier app.js pour définir les routes de l'API CRUD :

```
const express = require('express');
const bodyParser = require('body-parser');
const db = require('./db');
const app = express();
app.use(bodyParser.json());

// Route pour récupérer tous les éléments
app.get('/items', (req, res) => {
  db.query('SELECT * FROM items', (err, result) => {
    if (err) throw err;
    res.json(result);
  });
});

// Route pour récupérer un élément par son ID
app.get('/items/:id', (req, res) => {
  const itemId = req.params.id;
  db.query('SELECT * FROM items WHERE id = ?', itemId, (err, result)
=> {
    if (err) throw err;
    res.json(result[0]);
  });
});

// Route pour ajouter un nouvel élément
app.post('/items', (req, res) => {
  const newItem = req.body;
  db.query('INSERT INTO items SET ?', newItem, (err, result) => {
    if (err) throw err;
    res.send('Élément ajouté avec succès');
```

```
  });
});

// Route pour mettre à jour un élément par son ID
app.put('/items/:id', (req, res) => {
  const itemId = req.params.id;
  const updatedItem = req.body;
  db.query('UPDATE items SET ? WHERE id = ?', [updatedItem, ite-
mId], (err, result) => {
    if (err) throw err;
    res.send('Élément mis à jour avec succès');
  });
});

// Route pour supprimer un élément par son ID
app.delete('/items/:id', (req, res) => {
  const itemId = req.params.id;
  db.query('DELETE FROM items WHERE id = ?', itemId, (err, result)
=> {
    if (err) throw err;
    res.send('Élément supprimé avec succès');
  });
});

const PORT = 3000;
app.listen(PORT, () => {
  console.log(`Serveur en cours d'exécution sur le port ${PORT}`);
});
```

Étape 4 : Exécution de l'application

Dans votre terminal, exécutez le fichier app.js :

```
node app.js
```

Votre API CRUD Node.js avec MySQL est maintenant en cours d'exécution. Vous pouvez accéder aux différentes routes pour effectuer des opérations CRUD sur la table "items" de votre base de données MySQL.
Assurez-vous de remplacer les informations de connexion à la base de données

par les vôtres dans le fichier db.js.

Pour **ajouter une interface HTML à votre application Node.js** avec MySQL, vous pouvez utiliser des modèles de moteur de rendu comme **EJS (Embedded JavaScript)** ou Pug (anciennement Jade). Voici comment vous pouvez le faire en utilisant EJS :

Étape 1: Installer le moteur de rendu EJS

```
npm install ejs
```

Étape 2: Mettez à jour votre fichier app.js pour utiliser EJS comme moteur de rendu et ajoutez des routes pour les vues HTML.

```
const express = require('express');
const bodyParser = require('body-parser');
const db = require('./db');
const app = express();
app.use(bodyParser.json());

// Configuration du moteur de rendu EJS
app.set('view engine', 'ejs');
// Route pour afficher la liste des éléments
app.get('/items', (req, res) => {
  db.query('SELECT * FROM items', (err, result) => {
    if (err) throw err;
    res.render('items', { items: result });
  });
});

// Route pour afficher un élément par son ID
app.get('/items/:id', (req, res) => {
  const itemId = req.params.id;
  db.query('SELECT * FROM items WHERE id = ?', itemId, (err, result)
=> {
    if (err) throw err;
    res.render('item', { item: result[0] });
  });
```

```
});

// ... Les autres routes CRUD restent les mêmes
const PORT = 3000;
app.listen(PORT, () => {
  console.log(`Serveur en cours d'exécution sur le port ${PORT}`);
});
```

Étape 3: Créez les fichiers de modèle EJS

Créez un dossier views dans votre répertoire de projet.
À l'intérieur du dossier views, créez deux fichiers EJS: items.ejs et item.ejs.

items.ejs :

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, ini-
tial-scale=1.0">
  <title>Items</title>
</head>

<body>
  <h1>Liste des éléments</h1>
  <ul>
    <% items.forEach(function(item) { %>
      <li><a href="/items/<%= item.id %>"><%= item.name %></
a></li>
    <% }); %>
  </ul>
</body>
</html>
```

item.ejs :

```
<!DOCTYPE html>
```

```
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, ini-
tial-scale=1.0">
  <title><%= item.name %></title>
</head>

<body>
  <h1><%= item.name %></h1>
  <p>Description: <%= item.description %></p>
  <p>Prix: <%= item.price %></p>
</body>
</html>
```

Dans ces fichiers de modèle, <%= variable %> est utilisé pour insérer des valeurs de variables dans le HTML.

Avec ces étapes, votre application Node.js avec MySQL a maintenant une interface HTML basique. Lorsque vous accédez à http://localhost:3000/items, vous devriez voir la liste des éléments à partir de votre base de données. Assurez-vous de personnaliser les fichiers EJS selon vos besoins.

Coder un crud Node.js PostgreSQL

Pour créer un CRUD (Create, Read, Update, Delete) en Node.js avec PostgreSQL, vous aurez besoin des modules pg (pour la connexion à la base de données PostgreSQL) et express (pour créer l'API). Assurez-vous d'avoir Node.js installé sur votre machine avant de commencer. Voici un exemple simple de code pour un CRUD PostgreSQL en Node.js :

1. Installez les modules nécessaires :

```
npm install express pg
```

2. Créez un fichier app.js et ajoutez le code suivant :

```
const express = require('express');
const bodyParser = require('body-parser');
const { Client } = require('pg');
const app = express();
app.use(bodyParser.json());

// Configurer la connexion à la base de données PostgreSQL
const client = new Client({
  user: 'votre_utilisateur',
  host: 'votre_hôte',
  database: 'votre_base_de_données',
  password: 'votre_mot_de_passe',
  port: 'votre_port',
});
client.connect();

// Route pour récupérer tous les éléments
app.get('/api/elements', async (req, res) => {
```

```
  try {
    const result = await client.query('SELECT * FROM elements');
    res.json(result.rows);
  } catch (err) {
    console.error(err);
    res.status(500).json({ error: 'Erreur serveur' });
  }
});

// Route pour créer un nouvel élément
app.post('/api/elements', async (req, res) => {
  const { nom, description } = req.body;
  try {
    const result = await client.query('INSERT INTO elements(nom, des-
cription) VALUES($1, $2) RETURNING *', [nom, description]);
    res.json(result.rows[0]);
  } catch (err) {
    console.error(err);
    res.status(500).json({ error: 'Erreur serveur' });
  }
});

// Route pour mettre à jour un élément
app.put('/api/elements/:id', async (req, res) => {
  const { nom, description } = req.body;
  const id = req.params.id;
  try {
    const result = await client.query('UPDATE elements SET nom = $1,
description = $2 WHERE id = $3 RETURNING *', [nom, description,
id]);
    res.json(result.rows[0]);
  } catch (err) {
    console.error(err);
    res.status(500).json({ error: 'Erreur serveur' });
  }
});

// Route pour supprimer un élément
```

```
app.delete('/api/elements/:id', async (req, res) => {
  const id = req.params.id;
  try {
    await client.query('DELETE FROM elements WHERE id = $1', [id]);
    res.json({ message: 'Élément supprimé avec succès' });
  } catch (err) {
    console.error(err);
    res.status(500).json({ error: 'Erreur serveur' });
  }
});

const PORT = 3000;
app.listen(PORT, () => {
  console.log(`Serveur en écoute sur le port ${PORT}`);
});
```

Ce code crée un serveur Express qui expose quatre endpoints :

- GET /api/elements pour récupérer tous les éléments,
- POST /api/elements pour créer un nouvel élément,
- PUT /api/elements/:id pour mettre à jour un élément par son ID,
- DELETE /api/elements/:id pour supprimer un élément par son ID.

N'oubliez pas de remplacer les valeurs dans le bloc de configuration de la connexion PostgreSQL par les informations de votre base de données. Assurez-vous également de gérer les erreurs et les cas de bord pour un usage en production.

Pour intégrer le CRUD Node.js/PostgreSQL dans une page HTML, vous pouvez utiliser JavaScript côté client (AJAX) pour effectuer des requêtes vers votre API Node.js. Voici un exemple de code pour une page HTML avec des formulaires pour créer, mettre à jour et supprimer des éléments de votre base de données PostgreSQL :

1. Créez un fichier index.html :

```
<!DOCTYPE html>
<html lang="en">
  <head>
```

```
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, ini-
tial-scale=1.0">
    <title>CRUD Example</title>
  </head>
  <body>
    <h1>CRUD Example</h1>
    <h2>Ajouter un élément</h2>
    <form id="ajouterForm">
      <label for="nom">Nom:</label>
      <input type="text" id="nom" name="nom" required><br>
      <label for="description">Description:</label>
      <input type="text" id="description" name="description" require-
d><br>
      <button type="submit">Ajouter</button>
    </form>
    <h2>Liste des éléments</h2>
    <ul id="elementList"></ul>
    <script>
      // Fonction pour charger la liste des éléments
      function chargerElements() {
        fetch('/api/elements')
          .then(response => response.json())
          .then(data => {
            const elementList = document.getElementById('element-
List');
            elementList.innerHTML = '';
            data.forEach(element => {
              const li = document.createElement('li');
              li.textContent = `${element.nom}: ${element.descrip-
tion}`;
              elementList.appendChild(li);
            });
          })
          .catch(error => console.error('Erreur:', error));
      }
      // Écouter le formulaire d'ajout
      const ajouterForm = document.getElementById('ajouterForm');
```

```
        ajouterForm.addEventListener('submit', function (event) {
            event.preventDefault();
            const nom = document.getElementById('nom').value;
            const description = document.getElementById('description').
value;
            fetch('/api/elements', {
                method: 'POST',
                headers: {
                    'Content-Type': 'application/json'
                },
                body: JSON.stringify({
                    nom: nom,
                    description: description
                })
                .then(response => response.json())
                .then(data => {
                    console.log('Élément ajouté:', data);

                    // Recharger la liste des éléments après l'ajout
                    chargerElements();
                })
                .catch(error => console.error('Erreur:', error));
            });
            // Charger les éléments au chargement de la page
            chargerElements();
        </script>
    </body>
</html>
```

Dans ce code HTML, un formulaire permet d'ajouter un nouvel élément. La liste des éléments est chargée au chargement de la page et mise à jour après l'ajout d'un nouvel élément. Assurez-vous que le chemin de l'API ('/api/elements') correspond à l'endpoint de votre serveur Node.js.

N'oubliez pas de personnaliser ce code en fonction de vos besoins spécifiques et d'ajouter des fonctionnalités pour la mise à jour et la suppression d'éléments selon vos besoins.

Pour ajouter des fonctionnalités de mise à jour et de suppression d'éléments à votre application HTML, vous pouvez étendre le script JavaScript dans votre fichier index.html. Voici comment vous pouvez faire cela :

Ajouter la fonctionnalité de mise à jour :
Ajoutez un formulaire de mise à jour similaire à celui d'ajout, mais pré-rempli avec les détails de l'élément à mettre à jour. Vous pouvez utiliser un événement "click" sur chaque élément de la liste pour remplir automatiquement le formulaire de mise à jour. Lorsque le formulaire de mise à jour est soumis, effectuez une requête PUT à l'API pour mettre à jour l'élément.

```
<h2>Mettre à jour un élément</h2>
<form id="mettreAJourForm">
  <input type="hidden" id="elementId" name="id">
  <label for="nomMaj">Nom:</label>
  <input type="text" id="nomMaj" name="nom" required><br>
  <label for="descriptionMaj">Description:</label>
  <input type="text" id="descriptionMaj" name="description" require-
d><br>
  <button type="submit">Mettre à jour</button>
</form>

<script>
  // Écouter le clic sur un élément pour le mettre à jour
  function preparerMiseAJour(element) {
    document.getElementById('elementId').value = element.id;
    document.getElementById('nomMaj').value = element.nom;
    document.getElementById('descriptionMaj').value = element.des-
cription;
  }

  // Écouter le formulaire de mise à jour
  const mettreAJourForm = document.getElementById('mettreAJour-
Form');
  mettreAJourForm.addEventListener('submit', function (event) {
    event.preventDefault();
    const id = document.getElementById('elementId').value;
    const nom = document.getElementById('nomMaj').value;
```

```
    const description = document.getElementById('descriptionMaj').
value;
    fetch(`/api/elements/${id}`, {
      method: 'PUT',
      headers: {
        'Content-Type': 'application/json'
      },
        body: JSON.stringify({
          nom: nom,
           description: description
        })
      })
      .then(response => response.json())
      .then(data => {
        console.log('Élément mis à jour:', data);
        // Recharger la liste des éléments après la mise à jour
        chargerElements();
      })
      .catch(error => console.error('Erreur:', error));
    });
    // Ajouter un gestionnaire de clic aux éléments de la liste pour la mise
à jour
        document.getElementById('elementList').addEventListe-
ner('click', function (event) {
          if (event.target.tagName === 'LI') {
            // Récupérer l'ID de l'élément à mettre à jour
            const elementId = event.target.getAttribute('data-id');

            // Récupérer les détails de l'élément par son ID
            fetch(`/api/elements/${elementId}`)
            .then(response => response.json())
            .then(data => {

              // Pré-remplir le formulaire de mise à jour avec les détails
de l'élément
              preparerMiseAJour(data);
            })
            .catch(error => console.error('Erreur:', error));
```

```
        }
      });
    </script>
```

Ajouter la fonctionnalité de suppression :
Ajoutez un bouton de suppression à côté de chaque élément dans la liste. Écoutez le clic sur ce bouton et, lorsqu'il est cliqué, effectuez une requête DELETE à l'API pour supprimer l'élément correspondant.

```
<h2>Liste des éléments</h2>
<ul id="elementList"></ul>
<script>
  // Fonction pour charger la liste des éléments
  function chargerElements() {

  // ... (le code pour charger les éléments est inchangé)
}

// ... (le reste du code pour la mise à jour est inchangé)

// Ajouter un gestionnaire de clic aux éléments de la liste pour la suppression
document.getElementById('elementList').addEventListener('click', function (event) {
  if (event.target.tagName === 'BUTTON') {
    // Récupérer l'ID de l'élément à supprimer
    const elementId = event.target.getAttribute('data-id');
    // Supprimer l'élément par son ID
    fetch(`/api/elements/${elementId}`, {
      method: 'DELETE'
    })
    .then(response => response.json())
    .then(data => {
      console.log('Élément supprimé:', data);
        // Recharger la liste des éléments après la suppression
        chargerElements();
      })
      .catch(error => console.error('Erreur:', error));
```

```
    }
  });
  // ... (le reste du script est inchangé)
</script>
```

Avec ces modifications, votre application HTML dispose maintenant des fonctionnalités de mise à jour et de suppression des éléments. Assurez-vous que vos routes côté serveur (Node.js) sont correctement implémentées pour gérer les requêtes PUT (mise à jour) et DELETE (suppression).

Pour implémenter les routes côté serveur (Node.js) pour gérer les requêtes PUT (mise à jour) et DELETE (suppression), vous pouvez utiliser le framework Express. Voici comment vous pouvez mettre en œuvre ces routes dans votre serveur Node.js :

Mise à jour (Route PUT) :

```
// Route pour mettre à jour un élément par son ID
app.put('/api/elements/:id', async (req, res) => {
  const { nom, description } = req.body;
  const id = req.params.id;
  try {
    const result = await client.query('UPDATE elements SET nom = $1,
description = $2 WHERE id = $3 RETURNING *', [nom, description,
id]);
    if (result.rows.length === 0) {
      res.status(404).json({ error: 'Élément non trouvé' });
    } else {
      res.json(result.rows[0]);
    }
  } catch (err) {
    console.error(err);
    res.status(500).json({ error: 'Erreur serveur' });
  }
});
```

Suppression (Route DELETE) :

```
// Route pour supprimer un élément par son ID
app.delete('/api/elements/:id', async (req, res) => {
  const id = req.params.id;
  try {
    const result = await client.query('DELETE FROM elements WHERE
id = $1 RETURNING *', [id]);
    if (result.rows.length === 0) {
      res.status(404).json({ error: 'Élément non trouvé' });
    } else {
      res.json({ message: 'Élément supprimé avec succès' });
    }
  } catch (err) {
    console.error(err);
    res.status(500).json({ error: 'Erreur serveur' });
  }
});
```

Dans ces routes, :id est un paramètre dynamique dans l'URL qui correspond à l'identifiant de l'élément que vous souhaitez mettre à jour ou supprimer. Assurez-vous que ces routes sont définies après l'initialisation de votre client PostgreSQL et avant d'appeler app.listen(...). Ces routes géreront les requêtes PUT et DELETE envoyées depuis votre application frontend. Assurez-vous également que votre base de données PostgreSQL est correctement configurée et accessible depuis votre serveur Node.js.

Notez que dans un environnement de production, vous devez également gérer les erreurs de manière plus robuste, peut-être en utilisant un middleware Express pour gérer les erreurs.

Créer un client Webmail IMAP en Node.js

Créer un client Webmail IMAP en Node.js implique d'utiliser le protocole IMAP (Internet Message Access Protocol) pour se connecter à un serveur de messagerie et récupérer les e-mails.
Voici un exemple de base pour vous aider à commencer. Assurez-vous d'avoir les packages nécessaires installés en utilisant :

```
npm install express body-parser node-imap
```

Code JavaScript :

```
const express = require('express');
const bodyParser = require('body-parser');
const Imap = require('node-imap');
const simpleParser = require('mailparser').simpleParser;
const app = express();
app.use(bodyParser.json());

// Page d'accueil
app.get('/', (req, res) => {
  res.sendFile(__dirname + '/index.html');
});

// Endpoint pour récupérer les e-mails
app.get('/emails', (req, res) => {
  const imap = new Imap({
    user: 'votre_email@example.com',
    password: 'votre_mot_de_passe',
    host: 'imap.example.com',
    port: 993,
    tls: true
```

```
});

imap.connect();
imap.once('ready', () => {
imap.openBox('INBOX', true, (err, box) => {
  if (err) throw err;
  const fetch = imap.seq.fetch('1:*', {
    bodies: '',
    struct: true
  });
  fetch.on('message', (msg, seqno) => {
    const emailData = {
      seqno: seqno,
      body: ''
    };
    msg.on('body', (stream, info) => {
      let buffer = '';
      stream.on('data', chunk => {
        buffer += chunk.toString('utf8');
      });
      stream.once('end', () => {
        emailData.body = buffer;
      });
    });
    msg.once('attributes', attrs => {
      emailData.attributes = attrs;
    });
    msg.once('end', () => {
      simpleParser(emailData.body, (err, mail) => {
        emailData.parsedMail = mail;
        res.write(JSON.stringify(emailData) + '\n');
      });
    });
  });
  fetch.once('error', err => {
    console.log('Fetch error: ' + err);
  });
  fetch.once('end', () => {
```

```
            res.end();
            imap.end();
        });
    });
});
imap.once('error', err => {
    console.log('IMAP error: ' + err);
});
});
const PORT = 3000;
app.listen(PORT, () => {
    console.log(`Serveur en cours d'exécution sur le port ${PORT}`);
});
```

Dans cet exemple, nous utilisons le package node-imap pour se connecter à un serveur IMAP. Assurez-vous de remplacer les valeurs user, password, host et port avec les informations de votre compte de messagerie IMAP.

N'oubliez pas que stocker et manipuler les mots de passe en clair dans votre code n'est pas recommandé en production. Dans un environnement de production, vous devriez utiliser des variables d'environnement ou un fichier de configuration sécurisé pour stocker ces informations sensibles.
Ajouter la gestion des pièces jointes

Pour ajouter la gestion des pièces jointes aux e-mails récupérés à partir d'un serveur IMAP dans votre client Webmail Node.js, vous pouvez étendre le code précédent en utilisant la bibliothèque node-imap pour télécharger et gérer les pièces jointes. Voici comment vous pouvez faire cela :

```
const express = require('express');
const bodyParser = require('body-parser');
const Imap = require('node-imap');
const simpleParser = require('mailparser').simpleParser;
const fs = require('fs');
const app = express();
app.use(bodyParser.json());

// Page d'accueil
```

```
app.get('/', (req, res) => {
  res.sendFile(__dirname + '/index.html');
});

// Endpoint pour récupérer les e-mails avec pièces jointes
app.get('/emails', (req, res) => {
  const imap = new Imap({
    user: 'votre_email@example.com',
    password: 'votre_mot_de_passe',
    host: 'imap.example.com',
    port: 993,
    tls: true
  });
  imap.connect();
  imap.once('ready', () => {
    imap.openBox('INBOX', true, (err, box) => {
      if (err) throw err;
      imap.search(['ALL'], (err, results) => {
        if (err) throw err;
        const fetch = imap.fetch(results, { bodies: [''], struct: true });
        fetch.on('message', (msg, seqno) => {
          const emailData = {
            seqno: seqno,
            attachments: []
          };
          msg.on('body', (stream, info) => {
            let buffer = '';
            stream.on('data', chunk => {
              buffer += chunk.toString('utf8');
            });
            stream.once('end', () => {
              emailData.body = buffer;
            });
          });
          msg.on('attributes', attrs => {
            emailData.attributes = attrs;
          });
          msg.once('end', () => {
```

```
                simpleParser(emailData.body, (err, mail) => {
                    emailData.parsedMail = mail;

                    // Vérification des pièces jointes
                    if (mail.attachments && mail.attachments.length > 0) {
                        mail.attachments.forEach((attachment, index) => {
                            const filename = attachment.filename;
                            const filePath = `./attachments/${filename}`;
                            fs.writeFileSync(filePath, attachment.content);
                            emailData.attachments.push({
                                filename: filename,
                                path: filePath
                            });
                        });
                    }
                    res.write(JSON.stringify(emailData) + '\n');
                });
            });
        });
        fetch.once('error', err => {
            console.log('Fetch error: ' + err);
        });
        fetch.once('end', () => {
            res.end();
            imap.end();
        });
    });
});
imap.once('error', err => {
    console.log('IMAP error: ' + err);
});
const PORT = 3000;
app.listen(PORT, () => {
    console.log(`Serveur en cours d'exécution sur le port ${PORT}`);
});
```

Dans ce code, nous avons ajouté une section pour vérifier les pièces jointes dans l'e-mail et les enregistrer dans un dossier attachments. Assurez-vous que ce dossier existe dans le répertoire de votre application.
N'oubliez pas de personnaliser ce code en remplaçant les valeurs user, password, host, et port par vos propres informations de serveur IMAP. Ce code récupérera les e-mails, analysera leurs pièces jointes et les sauvegardera localement dans le dossier attachments. Vous pouvez ensuite ajouter la logique nécessaire pour afficher ou gérer ces pièces jointes dans votre application.

Comment utiliser EJS pour modéliser votre application Node.js

Introduction

Lors de la création rapide d'applications Node.js en temps réel, il est parfois nécessaire de disposer d'un moyen simple et rapide de modéliser notre application.

Jade est le moteur de visualisation d'Express par défaut, mais la syntaxe de Jade peut être trop complexe pour de nombreux cas d'utilisation. EJS est une alternative qui fait bien ce travail et qui est très facile à mettre en place. Voyons comment nous pouvons créer une application simple et utiliser EJS pour inclure des parties répétables de notre site (partiels) et transmettre des données à nos vues.

Mettre en place l'application de démonstration

Nous allons faire deux pages pour notre application : une page avec une page pleine et l'autre avec une barre latérale.

Structure de fichier

Voici les dossiers dont nous aurons besoin pour notre application. Nous allons faire notre modèle dans le dossier « views » et le reste est assez standard dans les pratiques de Node.js.

```
- views
----- partials
---------- footer.ejs
---------- head.ejs
```

```
---------- header.ejs
----- pages
---------- index.ejs
---------- about.ejs
- package.json
- server.js
```

package.json contiendra nos informations sur les applications Node.js et les dépendances dont nous avons besoin (express et EJS). server.js contiendra la configuration de notre serveur Express. Nous allons définir nos itinéraires vers nos pages ici.

Configuration Node.js

Allons dans notre package.json et nous y installerons notre projet.

package.json

```
{
  «name»: «node-ejs»,
  «main»: «server.js»,
  «dependencies»: {
    «ejs»: «^3.1.5»,
    «express»: «^4.17.1»
  }
}
```

Tout ce dont nous aurons besoin, c'est d'Express et EJS. Il nous faut maintenant installer les dépendances que nous venons de définir. Allez-y et exécutez :

```
npm install
```

Une fois toutes nos dépendances installées, configurons notre application pour utiliser EJS et définissons nos itinéraires pour les deux pages dont nous avons besoin : la page index (pleine largeur) et la page about (barre latérale). Nous ferons tout cela à l'intérieur de notre fichier server.js.

server.js

```
// load the things we need
var express = require('express');
var app = express();

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

// use res.render to load up an ejs view file

// index page
app.get('/', function(req, res) {
   res.render('pages/index');
});

// about page
app.get('/about', function(req, res) {
   res.render('pages/about');
});

app.listen(8080);
console.log('8080 is the magic port');
```

Ici, nous définissons notre application et nous la définissons pour qu'elle s'affiche sur le port 8080. Nous devons également définir EJS comme le moteur de visualisation de notre application Express en utilisant app.set('view engineer', 'ejs') ;. Remarquez comment nous envoyons une vue à l'utilisateur en utilisant res.render(). Il est important de noter que res.render() va chercher dans un dossier de vues pour le visualiser. Il nous suffit donc de définir pages/index puisque le chemin complet est views/pages/index.

Démarrez notre serveur

Allez-y et démarrez le serveur en utilisant :

```
node server.js
```

Nous pouvons maintenant voir notre application dans le navigateur à l'adresse

http://localhost:8080 et http://localhost:8080/about. Notre application est mise en place et nous devons définir nos fichiers de consultation et voir comment EJS fonctionne avec ceux-là.

Créez les fichiers partiels EJS

Comme beaucoup d'applications que nous construisons, il y aura beaucoup de code qui sera réutilisé. Nous appellerons ces partiels et définirons trois fichiers que nous utiliserons sur l'ensemble de notre site : head.ejs, header.ejs, et footer.ejs. Faisons maintenant ces fichiers.

views/partials/head.ejs

```
<meta charset=»UTF-8»>
<title>EJS Is Fun</title>
```

views/partials/header.ejs

```
<!-- CSS (load bootstrap from a CDN) -->
<link rel=»stylesheet» href=»https://cdnjs.cloudflare.com/ajax/libs/twitter-bootstrap/4.5.2/css/bootstrap.min.css»>
<style>
  body { padding-top:50px; }
</style>
```

views/partials/footer.ejs

```
<p class=»text-center text-muted»>© Copyright 2020 The Awesome People</p>
```

Vous remarquez que malgré l'extension .ejs, le code est bien du HTML.

Ajoutez les partiels EJS à Views

Nous avons maintenant défini nos partiels. Tout ce que nous avons à faire, c'est de les inclure dans nos vues. Allons dans index.ejs et about.ejs et utilisons la syntaxe include pour ajouter les partiels.

Syntaxe pour l'inclusion d'un partiel EJS

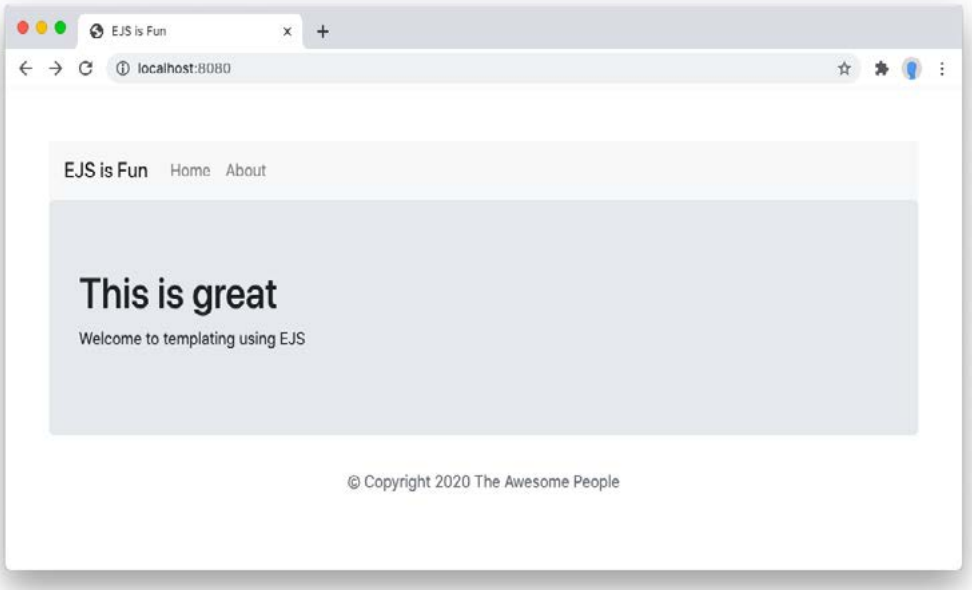
Utilisez `<%- include('RELATIVE/PATH/TO/FILE') %>` pour intégrer une partie de EJS dans un autre fichier.

Le trait d'union `<%-` au lieu de `<%` pour dire à EJS de rendre le HTML brut. Le chemin vers le partiel est relatif au fichier actuel.

views/pages/index.ejs

```
<!DOCTYPE html>
<html lang=»en»>
<head>
  <%- include('../partials/head'); %>
</head>
<body class=»container»>

<header>
  <%- include('../partials/header'); %>
</header>

<main>
  <div class=»jumbotron»>
    <h1>This is great</h1>
    <p>Welcome to templating using EJS</p>
  </div>
</main>

<footer>
  <%- include('../partials/footer'); %>
</footer>

</body>
</html>
```

Nous pouvons maintenant voir notre vue définie dans le navigateur à l'adresse http://localhost:8080



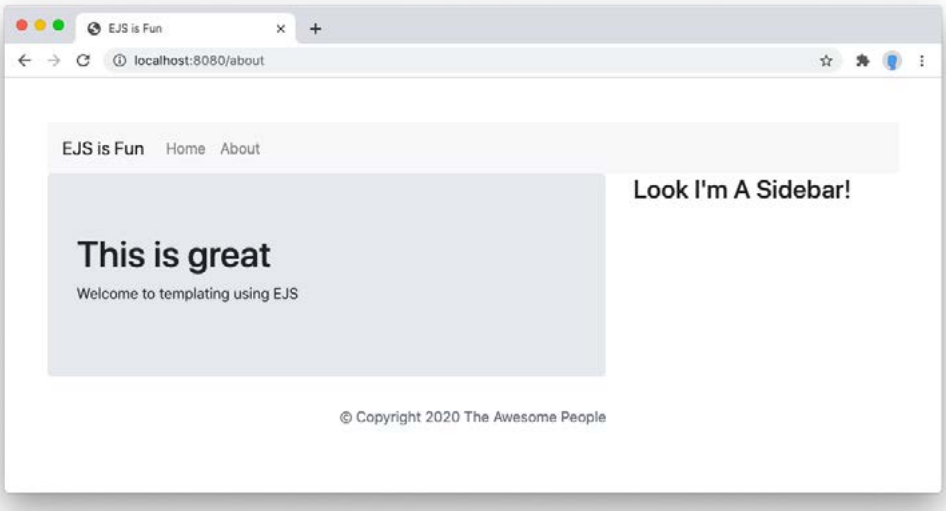
Pour la page about, nous ajoutons également une barre latérale bootstrap pour montrer comment les partiels peuvent être structurés pour être réutilisés dans différents modèles et pages.

views/pages/about.ejs

```
<!DOCTYPE html>
<html lang=»en»>
<head>
  <%- include('../partials/head'); %>
</head>
<body class=»container»>

<header>
  <%- include('../partials/header'); %>
</header>

<main>
<div class=»row»>
  <div class=»col-sm-8»>
    <div class=»jumbotron»>
```

```
        <h1>This is great</h1>
        <p>Welcome to templating using EJS</p>
      </div>
    </div>

    <div class=»col-sm-4»>
      <div class=»well»>
        <h3>Look I'm A Sidebar!</h3>
      </div>
    </div>

  </div>
  </main>

  <footer>
    <%- include('../partials/footer'); %>
  </footer>

  </body>
  </html>
```

Si nous visitons http://localhost:8080/about nous pouvons consulter notre page about avec une barre latérale !



Nous pouvons maintenant commencer à utiliser EJS pour transmettre les données de notre application Node.js à nos vues.

Transmettre les données aux vues et aux partiels

Définissons quelques variables de base et une liste à transmettre à notre page d'accueil. Retournez dans votre fichier server.js et ajoutez ce qui suit dans votre itinéraireapp.get('/').

server.js

```
// index page
app.get('/', function(req, res) {
  var mascots = [
    { name: 'Sammy', organization: «DigitalOcean», birth_year: 2012},
    { name: 'Tux', organization: «Linux», birth_year: 1996},
    { name: 'Moby Dock', organization: «Docker», birth_year: 2013}
  ];
  var tagline = «No programming concept is complete without a cute
animal mascot.»;

  res.render('pages/index', {
    mascots: mascots,
    tagline: tagline
  });
});
```

Nous avons créé une liste appelée mascots et une chaîne simple appelée tagline. Allons dans notre fichier index.ejs et utilisons-les.

Rendre une variable unique dans EJS

Pour faire écho à une seule variable, nous utilisons simplement `<%= tagline %>`. Ajoutons ceci à notre fichier index.ejs :

views/pages/index.ejs

```
...
<h2>Variable</h2>
<p><%= tagline %></p>
...
```

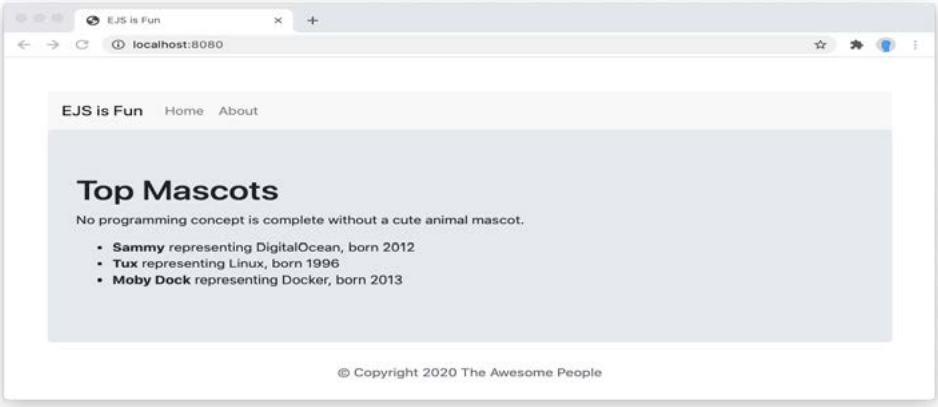
Boucler les données dans EJS

Pour boucler nos données, nous utiliserons .forEach. Ajoutons ceci à notre nouveau fichier :

views/pages/index.ejs

```
...
<ul>
  <% mascots.forEach(function(mascot) { %>
    <li>
      <strong><%= mascot.name %></strong>
      representing <%= mascot.organization %>, born <%= mascot.
birth_year %>
    </li>
  <% }); %>
</ul>
...
```

Nous pouvons maintenant voir dans notre navigateur les nouvelles informations que nous avons ajoutées !



Transmettre les données à un partiel dans EJS

Le partiel EJS a accès à toutes les données identiques à celles de la vue parente. Mais attention : Si vous faites référence à une variable dans un partiel, elle doit être définie dans chaque vue qui utilise le partiel ou elle lancera une erreur.

Vous pouvez également définir et transmettre des variables à un partiel EJS dans la syntaxe include, comme ceci :

views/pages/about.ejs

```
...
<header>
  <%- include('../partials/header', {variant:'compact'}); %>
</header>
...
```

Mais vous devez encore une fois faire attention à ne pas supposer qu'une variable a été définie.

Si vous souhaitez référencer une variable dans un partiel qui n'est pas toujours défini, et lui donner une valeur par défaut, vous pouvez le faire de cette manière :

views/partials/header.ejs

```
...
<em>Variant: <%= typeof variant != 'undefined' ? variant : 'default'
%></em>
...
```

Dans la ligne ci-dessus, le code EJS rend la valeur de variant si elle est définie, et par default si ce n'est pas le cas.

Conclusion

EJS nous permet de créer des applications rapides lorsque nous n'avons pas besoin de quelque chose de trop complexe. En utilisant des partiels et en ayant

la possibilité de passer facilement des variables à nos vues, nous pouvons construire rapidement d'excellentes applications.

Pour plus de références sur lEJS, voir le les documents officiels ici : https://ejs.co/#docs

Créer logiciel de gestion des bénévoles avec planning en node.js, express, ejs, mysql

Pour créer un logiciel de gestion des bénévoles avec un planning en utilisant Node.js, Express, EJS et MySQL, vous pouvez suivre ces étapes :

Étape 1: Installer les dépendances

```
npm install express ejs mysql body-parser
```

Étape 2: Configurer la connexion à la base de données MySQL

```
const mysql = require('mysql');
const connection = mysql.createConnection({
  host: 'votre_hôte_mysql',
  user: 'votre_nom_utilisateur',
  password: 'votre_mot_de_passe',
  database: 'votre_base_de_donnees'
});

connection.connect((err) => {
  if (err) {
    console.error('Erreur de connexion à la base de données : ' + err.
stack);
    return;
  }
  console.log('Connecté à la base de données MySQL');
});

module.exports = connection;
```

Étape 3: Configurer le serveur Express avec EJS

```
const express = require('express');
const bodyParser = require('body-parser');
const db = require('./db'); // Importez votre connexion à MySQL

const app = express();
const PORT = 3000;

app.set('view engine', 'ejs');
app.use(bodyParser.urlencoded({ extended: true }));

// Routes
app.get('/', (req, res) => {
  // Logique pour récupérer les données des bénévoles depuis la base de
données
  // Logique pour récupérer les données du planning depuis la base de
données
  db.query('SELECT * FROM benevoles', (err, benevoles) => {
    if (err) throw err;

    db.query('SELECT * FROM planning', (err, planning) => {
      if (err) throw err;

      res.render('index', { benevoles: benevoles, planning: planning });
    });
  });
});

app.post('/ajouter-benevole', (req, res) => {
  const nom = req.body.nom;
  // Logique pour ajouter un nouveau bénévole dans la base de données
  db.query('INSERT INTO benevoles (nom) VALUES (?)', [nom], (err,
result) => {
    if (err) throw err;

    res.redirect('/');
  });
```

```
});

app.post('/ajouter-evenement', (req, res) => {
  const nouvelEvenement = req.body.nouvelEvenement;
  // Logique pour ajouter un nouvel événement dans la base de données
  db.query('INSERT INTO planning (event) VALUES (?)', [nouvelEvene-
ment], (err, result) => {
    if (err) throw err;

    res.redirect('/');
  });
});

app.listen(PORT, () => {
  console.log(`Serveur en cours d'exécution sur le port ${PORT}`);
});
```

Étape 4: Créer les vues EJS pour l'interface utilisateur

Créez views/index.ejs pour afficher les bénévoles et le planning :

```
<!DOCTYPE html>
<html lang=»en»>

<head>
  <meta charset=»UTF-8»>
  <meta name=»viewport» content=»width=device-width, ini-
tial-scale=1.0»>
  <title>Gestion des Bénévoles</title>
</head>

<body>
  <h1>Liste des Bénévoles</h1>
  <ul>
    <% benevoles.forEach(benevole => { %>
      <li><%= benevole.nom %></li>
    <% }); %>
  </ul>
```

```
    <h2>Ajouter un Bénévole</h2>
    <form action="/ajouter-benevole" method="post">
      <input type="text" name="nom" placeholder="Nom du Bénévole"
required>
      <button type="submit">Ajouter</button>
    </form>

    <h1>Planning</h1>
    <ul>
      <% planning.forEach(event => { %>
        <li><%= event.event %></li>
      <% }); %>
    </ul>

    <h2>Ajouter un Événement</h2>
    <form action="/ajouter-evenement" method="post">
      <input type="text" name="nouvelEvenement" placeholder="Nouvel
Événement" required>
      <button type="submit">Ajouter</button>
    </form>
</body>

</html>
```

Assurez-vous que votre structure de dossier ressemble à ceci :

```
- mon_projet
  - node_modules
  - views
    - index.ejs
  - db.js
  - server.js
  - package.json
```

Dans cet exemple, vous avez une interface utilisateur qui affiche les bénévoles et le planning. Vous pouvez ajouter de nouveaux bénévoles et événements en utilisant les formulaires.

Assurez-vous de personnaliser le code pour correspondre à votre schéma de base de données et aux exigences spécifiques de votre application. N'oubliez pas de gérer les erreurs et d'ajouter des fonctionnalités de validation des formulaires pour une meilleure expérience utilisateur.

Pour **affecter des bénévoles à un événement** dans votre application de gestion des bénévoles avec Node.js, Express, EJS et MySQL, vous pouvez ajouter une nouvelle table dans votre base de données pour gérer les affectations. Voici comment vous pouvez le faire :

Étape 1: Mettez à jour votre base de données

Créez une nouvelle table dans votre base de données MySQL pour gérer les affectations entre les bénévoles et les événements. Voici un exemple de schéma de table :

```
CREATE TABLE affectations (
  id INT AUTO_INCREMENT PRIMARY KEY,
  benevole_id INT,
  evenement_id INT,
  FOREIGN KEY (benevole_id) REFERENCES benevoles(id),
  FOREIGN KEY (evenement_id) REFERENCES planning(id)
);
```

Dans cette table affectations, benevole_id fait référence à l'ID d'un bénévole dans la table benevoles, et evenement_id fait référence à l'ID d'un événement dans la table planning.

Étape 2: Mettez à jour votre modèle de données

Mettez à jour votre modèle de données pour refléter la nouvelle structure de la base de données. Vous pouvez créer de nouveaux modèles ou étendre les modèles existants pour inclure les affectations.

```
// Modèle de bénévole
const Benevole = {
  // ...
};
```

```
// Modèle d'événement
const Evenement = {
  // ...
};

// Modèle d'affectation
const Affectation = {
  // ...
};
```

Étape 3: Mettez à jour vos routes et vos vues

Mettez à jour votre route pour afficher les événements et les bénévoles :

```
app.get('/', (req, res) => {
  // Logique pour récupérer les données des bénévoles depuis la base de
données
  // Logique pour récupérer les données du planning depuis la base de
données
  // Logique pour récupérer les affectations depuis la base de données

  res.render('index', { benevoles: benevoles, planning: planning, affecta-
tions: affectations });
});
```

Mettez à jour votre vue EJS pour afficher les bénévoles et les événements, et ajoutez une option pour affecter les bénévoles aux événements :

```
<h1>Liste des Bénévoles</h1>
<ul>
  <% benevoles.forEach(benevole => { %>
    <li>
      <%= benevole.nom %>
      <select name=»evenement»>
        <% planning.forEach(evenement => { %>
          <option value=»<%= evenement.id %>»> <%= evenement.
```

```
event %> </option>
            <% }); %>
        </select>
        <button onclick=»affecterBenevole(‘<%= benevole.id %>’)»>Affec-
ter à l’événement</button>
    </li>
  <% }); %>
</ul>
```

Ajoutez un script JavaScript dans votre vue pour envoyer les affectations au serveur via une requête AJAX :

```
<script>
  function affecterBenevole(benevoleId) {
    const evenementId = document.querySelector(`[name=»evene-
ment»]`).value;

    // Envoie de l’affectation au serveur
    fetch(`/affecter-benevole/${benevoleId}/${evenementId}`, {
      method: ‘POST’,
    })
    .then(response => response.json())
    .then(data => {
      // Mettez à jour l’interface utilisateur en conséquence, si nécessaire
    })
    .catch(error => console.error(error));
  }
</script>
```

Mettez à jour votre route pour gérer l’affectation des bénévoles aux événements :

```
app.post(‘/affecter-benevole/:benevoleId/:evenementId’, (req, res) => {
  const benevoleId = req.params.benevoleId;
  const evenementId = req.params.evenementId;

  // Logique pour enregistrer l’affectation dans la base de données
  db.query(‘INSERT INTO affectations (benevole_id, evenement_id)
```

```
VALUES (?, ?)', [benevoleId, evenementId], (err, result) => {
    if (err) {
        res.status(500).json({ message: 'Erreur lors de l\'affectation du
bénévole à l\'événement.' });
    } else {
        res.status(200).json({ message: 'Bénévole affecté à l\'événement
avec succès.' });
    }
  });
});
```

Assurez-vous d'ajuster la logique de l'affectation des bénévoles aux événements selon vos besoins spécifiques. Ce processus implique l'utilisation de requêtes AJAX pour envoyer les données d'affectation du client au serveur de manière asynchrone.