1. Introduction à Express.js
   * Présentation d'Express.js et son utilité
   * Relation entre Node.js et Express.js
   * Installation d'Express.js
2. Première application Express
   * Création d'un projet Express
   * Structure d'une application Express
   * Création d'un serveur HTTP simple
3. Routage avec Express.js
   * Présentation des routes
   * Création de routes simples (GET, POST, PUT, DELETE)
   * Gestion des paramètres de route et de requête
4. Middleware
   * Explication du concept de middleware
   * Utilisation de middleware intégré (ex. express.json(), express.urlencoded())
   * Création de middleware personnalisé
5. Gestion des erreurs
   * Gestion des erreurs avec middleware
   * Création d'erreurs personnalisées
   * Gestion des erreurs 404 (Page non trouvée)
6. Intégration de moteurs de templates
   * Présentation des moteurs de templates (EJS, Pug, Handlebars, etc.)
   * Configuration et utilisation d'un moteur de templates
   * Création de vues et utilisation de données dynamiques
7. Travailler avec des données
   * Introduction aux bases de données (SQL, NoSQL)
   * Utilisation de modules pour se connecter aux bases de données (ex. Mongoose pour MongoDB)
   * Création de modèles de données et exécution de requêtes CRUD
8. Authentification et autorisation
   * Concepts d'authentification et d'autorisation
   * Utilisation de Passport.js pour l'authentification
   * Protection des routes et autorisation basée sur les rôles
9. Déploiement d'une application Express
   * Préparation de l'application pour le déploiement
   * Options de déploiement (Heroku, AWS, etc.)
   * Processus de déploiement sur une plateforme spécifique
10. Bonnes pratiques et astuces
    * Organisation du code et modularisation
    * Gestion des variables d'environnement
    * Conseils de performance et de sécurité

**1. Introduction à Express.js**

Express.js est un framework web minimaliste et flexible pour Node.js, conçu pour faciliter le développement d'applications web et d'API. Il fournit des outils et des fonctionnalités pour simplifier la gestion des routes, des requêtes, des réponses et des middlewares, permettant ainsi aux développeurs de se concentrer sur la logique métier de leur application.

Présentation d'Express.js et son utilité

Express.js est particulièrement utile pour :

* Gérer facilement les routes et les requêtes HTTP (GET, POST, PUT, DELETE, etc.)
* Gérer les middlewares pour effectuer des tâches communes, telles que la journalisation, l'authentification, le traitement des erreurs, etc.
* Simplifier la gestion des réponses HTTP, avec des méthodes pour définir les codes de statut, les en-têtes et le contenu de la réponse
* Faciliter la création d'API RESTful
* Intégrer divers moteurs de rendu de vues pour créer des applications web dynamiques

**Relation entre Node.js et Express.js**

Node.js est une plateforme open-source basée sur le moteur JavaScript V8 de Google Chrome, qui permet d'exécuter du JavaScript côté serveur. Node.js offre des fonctionnalités de bas niveau pour la création de serveurs HTTP, le traitement des requêtes et la gestion des événements.

Express.js est un framework construit au-dessus de Node.js pour simplifier le développement d'applications web et d'API. Il étend les fonctionnalités de base de Node.js en fournissant des outils et des fonctionnalités de haut niveau pour faciliter les tâches courantes du développement web.

**Installation d'Express.js**

Pour installer Express.js, vous devez d'abord installer Node.js et npm (le gestionnaire de paquets Node.js) sur votre système. Une fois Node.js et npm installés, vous pouvez installer Express.js en exécutant la commande suivante dans votre terminal ou invite de commande :

bashCopy code

npm install express

Cela installera Express.js dans votre projet et ajoutera une dépendance dans votre fichier **package.json**. Vous pouvez maintenant commencer à utiliser Express.js pour développer votre application web ou API.

**2. Première application Express**

Dans cette section, nous aborderons la création d'un projet Express, la structure d'une application Express et la création d'un serveur HTTP simple.

**2.1 Création d'un projet Express**

Pour créer un nouveau projet Express, suivez les étapes suivantes :

1. Ouvrez un terminal et créez un nouveau dossier pour votre projet :

bashCopy code

mkdir mon\_projet\_express cd mon\_projet\_express

1. Initialisez un nouveau projet Node.js en exécutant :

csharpCopy code

npm init -y

1. Installez Express.js en tant que dépendance du projet :

Copy code

npm install express

**2.2 Structure d'une application Express**

La structure minimale d'une application Express inclut un fichier JavaScript principal (par exemple **app.js**) et le fichier **package.json** généré lors de l'initialisation du projet. Le fichier JavaScript principal contient le code pour configurer et exécuter le serveur Express.

Voici un exemple de fichier **app.js** :

javascriptCopy code

const express = require('express'); const app = express(); const port = 3000; app.get('/', (req, res) => { res.send('Bienvenue sur notre serveur Express.js !'); }); app.listen(port, () => { console.log(`Serveur en écoute sur le port ${port}`); });

**2.3 Création d'un serveur HTTP simple**

Pour créer un serveur HTTP simple avec Express, suivez ces étapes :

1. Ouvrez le fichier **app.js** (ou un autre nom que vous avez choisi) dans votre éditeur de texte préféré.
2. Importez le module Express et créez une instance de l'application Express :

javascriptCopy code

const express = require('express'); const app = express();

1. Définissez le port d'écoute pour votre serveur :

javascriptCopy code

const port = 3000;

1. Créez une route pour la racine (GET sur "/") qui envoie un message de bienvenue :

javascriptCopy code

app.get('/', (req, res) => { res.send('Bienvenue sur notre serveur Express.js !'); });

1. Démarrez le serveur et écoutez sur le port spécifié :

javascriptCopy code

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

1. Exécutez votre application avec la commande suivante dans le terminal :

Copy code

node app.js

Votre serveur HTTP simple est maintenant en cours d'exécution et écoute sur le port 3000. Vous pouvez accéder à votre application en ouvrant un navigateur et en accédant à l'URL **http://localhost:3000**.

Haut du formulaire

## Routage avec Express.js

Le routage est un mécanisme essentiel pour gérer les requêtes HTTP entrantes et orienter ces requêtes vers le bon gestionnaire de route en fonction de l'URL et de la méthode HTTP utilisée. Express.js offre un système de routage simple et puissant pour gérer ces requêtes.

### Présentation des routes

Les routes définissent les points d'entrée pour votre application et déterminent comment celle-ci doit répondre aux requêtes HTTP entrantes. Express.js prend en charge toutes les méthodes HTTP standard, telles que GET, POST, PUT et DELETE, pour définir des routes.

### Création de routes simples (GET, POST, PUT, DELETE)

Voici comment définir des routes simples avec les méthodes GET, POST, PUT et DELETE :

javascriptCopy code

const express = require('express'); const app = express(); // Route GET app.get('/', (req, res) => { res.send('Requête GET sur la racine'); }); // Route POST app.post('/ajouter', (req, res) => { res.send('Requête POST sur /ajouter'); }); // Route PUT app.put('/modifier', (req, res) => { res.send('Requête PUT sur /modifier'); }); // Route DELETE app.delete('/supprimer', (req, res) => { res.send('Requête DELETE sur /supprimer'); }); app.listen(3000, () => { console.log('Serveur en écoute sur le port 3000'); });

### Gestion des paramètres de route et de requête

Express.js vous permet de capturer les paramètres de route et les paramètres de requête pour personnaliser les réponses en fonction des données fournies par le client.

#### **Paramètres de route**

Les paramètres de route sont définis en utilisant **:param** dans la définition de la route et sont accessibles via l'objet **req.params**.

javascriptCopy code

app.get('/utilisateurs/:id', (req, res) => { res.send(`Utilisateur avec l'ID: ${req.params.id}`); });

#### **Paramètres de requête**

Les paramètres de requête sont fournis dans l'URL après le symbole **?** et sont accessibles via l'objet **req.query**.

javascriptCopy code

app.get('/recherche', (req, res) => { const termeRecherche = req.query.q; res.send(`Résultats de la recherche pour : ${termeRecherche}`); });

En combinant ces deux types de paramètres, vous pouvez créer des routes plus complexes et personnalisées pour répondre aux besoins spécifiques de votre application.

1. Middleware

Le middleware est un concept important dans les applications Express.js. Il s'agit de fonctions intermédiaires qui peuvent exécuter du code, modifier les objets de requête et de réponse, ou mettre fin au cycle de requête-réponse. Les middlewares sont souvent utilisés pour ajouter des fonctionnalités ou gérer des aspects spécifiques de la requête avant d'atteindre la route finale.

• Explication du concept de middleware :

Dans une application Express, le middleware est une série de fonctions qui sont exécutées dans l'ordre où elles sont définies. Chaque fonction middleware a accès aux objets de requête (req), de réponse (res) et à la fonction next(). La fonction next() est utilisée pour passer le contrôle au middleware suivant dans la chaîne.

• Utilisation de middleware intégré (ex. express.json(), express.urlencoded()) :

Express.js fournit quelques middlewares intégrés pour des tâches courantes. Voici deux exemples :

1. express.json() : Ce middleware est utilisé pour analyser les corps de requête au format JSON. Il rend les données JSON disponibles sous forme d'objet JavaScript dans req.body.

javascriptCopy code

const express = require("express"); const app = express(); app.use(express.json());

1. express.urlencoded() : Ce middleware est utilisé pour analyser les corps de requête encodés au format "x-www-form-urlencoded". Il rend les données de formulaire disponibles sous forme d'objet JavaScript dans req.body.

javascriptCopy code

const express = require("express"); const app = express(); app.use(express.urlencoded({ extended: false }));

• Création de middleware personnalisé :

Vous pouvez créer votre propre middleware pour effectuer des actions spécifiques dans votre application. Par exemple, vous pouvez créer un middleware pour enregistrer les informations de chaque requête dans un fichier journal.

javascriptCopy code

const express = require("express"); const app = express(); const logMiddleware = (req, res, next) => { console.log(`${req.method} ${req.url}`); next(); }; app.use(logMiddleware); app.get("/", (req, res) => { res.send("Accueil"); }); app.listen(3000, () => { console.log("Serveur en écoute sur le port 3000"); });

Dans cet exemple, la fonction logMiddleware est un middleware personnalisé qui enregistre la méthode et l'URL de chaque requête. La fonction next() est appelée pour passer le contrôle au middleware ou à la route suivante.

1. Gestion des erreurs

Dans une application Express.js, la gestion des erreurs est essentielle pour assurer la robustesse et la stabilité de l'application. Express.js permet de gérer les erreurs à l'aide de middleware et d'erreurs personnalisées.

• Gestion des erreurs avec middleware :

Un middleware d'erreur est une fonction qui prend quatre arguments : (err, req, res, next). Il est utilisé pour intercepter et gérer les erreurs dans l'application. Lorsqu'une erreur est détectée, elle est passée au middleware d'erreur à l'aide de la fonction next().

javascriptCopy code

const express = require("express"); const app = express(); app.get("/", (req, res, next) => { // Simule une erreur const error = new Error("Quelque chose s'est mal passé"); next(error); }); const errorHandler = (err, req, res, next) => { console.error(err.message); res.status(500).send("Une erreur s'est produite"); }; app.use(errorHandler); app.listen(3000, () => { console.log("Serveur en écoute sur le port 3000"); });

• Création d'erreurs personnalisées :

Vous pouvez créer des erreurs personnalisées en étendant la classe Error native de JavaScript. Les erreurs personnalisées peuvent être utilisées pour fournir des messages d'erreur plus détaillés ou pour gérer des erreurs spécifiques à votre application.

javascriptCopy code

class CustomError extends Error { constructor(message, statusCode) { super(message); this.statusCode = statusCode; } }

• Gestion des erreurs 404 (Page non trouvée) :

Pour gérer les erreurs 404, vous pouvez ajouter un middleware à la fin de la chaîne de middleware et de routage pour intercepter les requêtes qui n'ont pas été gérées par les routes précédentes.

javascriptCopy code

const express = require("express"); const app = express(); app.get("/", (req, res) => { res.send("Accueil"); }); // Middleware pour gérer les erreurs 404 app.use((req, res, next) => { res.status(404).send("Page non trouvée"); }); app.listen(3000, () => { console.log("Serveur en écoute sur le port 3000"); });

Dans cet exemple, le middleware pour les erreurs 404 est ajouté après la route principale. Si aucune route précédente ne correspond à la requête, le middleware enverra une réponse avec le statut 404 et le message "Page non trouvée".

1. Intégration de moteurs de templates

• Présentation des moteurs de templates (EJS, Pug, Handlebars, etc.) :

Les moteurs de templates sont des outils qui permettent de générer du HTML à partir de fichiers de modèle et de données. Ils facilitent la séparation de la logique de l'application et la présentation, rendant le code plus maintenable et lisible. Express.js est compatible avec plusieurs moteurs de templates tels que EJS, Pug, Handlebars, etc.

• Configuration et utilisation d'un moteur de templates :

consultez la documentation officielle EJS : [https://ejs.co](https://ejs.co/)

Pour utiliser un moteur de templates avec Express.js, vous devez d'abord installer le package correspondant via npm. Ensuite, vous devez configurer Express.js pour utiliser ce moteur de templates en définissant les options "view engine" et "views" dans votre application.

Par exemple, pour utiliser EJS, vous feriez :

javascriptCopy code

const express = require("express"); const app = express(); // Installer EJS avec `npm install ejs` app.set("view engine", "ejs"); app.set("views", "./views"); // Routes et logique de l'application app.listen(3000, () => { console.log("Serveur en écoute sur le port 3000"); });

• Création de vues et utilisation de données dynamiques :

Une fois le moteur de templates configuré, vous pouvez créer des fichiers de vue dans le dossier spécifié (par exemple, le dossier "views" dans l'exemple précédent). Les fichiers de vue sont écrits en utilisant la syntaxe du moteur de templates choisi.

Dans une route, vous pouvez utiliser la méthode **res.render()** pour afficher une vue et passer des données dynamiques. Par exemple, avec EJS :

1. Créez un fichier **views/home.ejs** avec le contenu suivant :

htmlCopy code

<!DOCTYPE html> <html lang="en"> <head> <meta charset="UTF-8"> <meta name="viewport" content="width=device-width, initial-scale=1.0"> <title>Accueil</title> </head> <body> <h1>Bienvenue, <%= name %> !</h1> </body> </html>

1. Ajoutez une route dans votre fichier JavaScript :

javascriptCopy code

app.get("/", (req, res) => { res.render("home", { name: "Utilisateur" }); });

Dans cet exemple, la méthode **res.render()** utilise le fichier **home.ejs** et lui transmet un objet contenant la propriété **name**. La syntaxe **<%= name %>** dans le fichier EJS permet d'insérer la valeur de la variable **name** dans le HTML généré.

1. Travailler avec des données

• Introduction aux bases de données (SQL, NoSQL) :

Les bases de données sont des systèmes de stockage et de gestion de données structurées, permettant aux applications de stocker, récupérer et manipuler des informations. Il existe deux types principaux de bases de données : les bases de données SQL (relationnelles) et les bases de données NoSQL (non relationnelles).

Les bases de données SQL, telles que MySQL, PostgreSQL et SQL Server, utilisent un langage structuré de requêtes (SQL) pour interagir avec les données. Elles sont basées sur des schémas rigides et des relations entre les tables.

Les bases de données NoSQL, telles que MongoDB, CouchDB et Cassandra, n'utilisent pas de langage de requête structuré comme SQL. Elles sont conçues pour être plus flexibles et scalables, et sont souvent utilisées dans des applications nécessitant un stockage de données évolutif ou des structures de données moins rigides.

• Utilisation de modules pour se connecter aux bases de données (ex. Mongoose pour MongoDB) :

Pour interagir avec une base de données dans une application Express.js, vous pouvez utiliser des modules spécifiques. Par exemple, pour se connecter à MongoDB, on utilise généralement Mongoose, un ODM (Object Data Mapper) qui facilite la gestion des données.

Vous pouvez installer Mongoose via npm :

Copy code

npm install mongoose

Ensuite, vous pouvez utiliser Mongoose pour vous connecter à votre base de données MongoDB et définir des modèles pour vos collections.

• Création de modèles de données et exécution de requêtes CRUD :

Avec Mongoose, vous pouvez définir des modèles qui représentent les collections dans votre base de données MongoDB. Les modèles permettent de définir des schémas pour la structure des données et fournissent des méthodes pour effectuer des opérations CRUD (Create, Read, Update, Delete) sur les collections.

Voici un exemple de création d'un modèle simple pour un utilisateur avec Mongoose :

javascriptCopy code

const mongoose = require("mongoose"); const userSchema = new mongoose.Schema({ name: String, email: String, age: Number, }); const User = mongoose.model("User", userSchema);

Une fois le modèle créé, vous pouvez utiliser les méthodes de Mongoose pour interagir avec la collection d'utilisateurs dans votre base de données, par exemple :

javascriptCopy code

// Créer un nouvel utilisateur const newUser = new User({ name: "John Doe", email: "john@example.com", age: 30 }); await newUser.save(); // Rechercher un utilisateur par email const user = await User.findOne({ email: "john@example.com" }); // Mettre à jour l'âge d'un utilisateur await User.updateOne({ \_id: user.\_id }, { age: 31 }); // Supprimer un utilisateur await User.deleteOne({ \_id: user.\_id });

Ces exemples illustrent comment utiliser Mongoose pour effectuer des opérations CRUD de base sur les données dans une application Express.js.

1. Authentification et autorisation

• Concepts d'authentification et d'autorisation :

L'authentification et l'autorisation sont deux concepts importants dans la sécurité des applications web. L'authentification consiste à vérifier l'identité d'un utilisateur, généralement en vérifiant ses identifiants, tels que le nom d'utilisateur et le mot de passe. L'autorisation, quant à elle, consiste à déterminer les droits d'accès et les permissions d'un utilisateur authentifié, par exemple, en déterminant quelles routes ou ressources il peut accéder.

• Utilisation de Passport.js pour l'authentification :

Passport.js est un middleware d'authentification pour Node.js et Express qui simplifie la gestion de l'authentification dans les applications web. Passport.js prend en charge de nombreuses stratégies d'authentification, y compris les authentifications locales (utilisateur/mot de passe), OAuth, et d'autres fournisseurs d'identité tels que Google, Facebook et Twitter.

Pour utiliser Passport.js, installez-le via npm :

Copy code

npm install passport

Vous pouvez ensuite configurer Passport.js avec la stratégie d'authentification souhaitée et l'intégrer dans votre application Express.js.

• Protection des routes et autorisation basée sur les rôles :

Une fois l'authentification mise en place, vous pouvez protéger les routes de votre application en vérifiant si un utilisateur est authentifié et s'il possède les autorisations appropriées pour accéder à une route spécifique. Vous pouvez le faire en créant un middleware personnalisé qui vérifie l'authentification et les autorisations de l'utilisateur.

Voici un exemple de middleware simple pour vérifier si un utilisateur est authentifié :

javascriptCopy code

function isAuthenticated(req, res, next) { if (req.user) { next(); } else { res.status(401).send("Accès non autorisé"); } }

Vous pouvez utiliser ce middleware pour protéger les routes de votre application :

javascriptCopy code

app.get("/protected", isAuthenticated, (req, res) => { res.send("Accès autorisé à la route protégée"); });

Pour une autorisation basée sur les rôles, vous pouvez étendre le middleware pour vérifier si l'utilisateur authentifié possède le rôle requis pour accéder à la route :

javascriptCopy code

function hasRole(role) { return (req, res, next) => { if (req.user && req.user.role === role) { next(); } else { res.status(403).send("Accès interdit"); } }; }

Ensuite, vous pouvez utiliser ce middleware pour protéger les routes en fonction des rôles :

javascriptCopy code

app.get("/admin", isAuthenticated, hasRole("admin"), (req, res) => { res.send("Accès autorisé à la route réservée aux administrateurs"); });

Ces exemples montrent comment gérer l'authentification et l'autorisation dans une application Express.js en utilisant Passport.js et des middlewares personnalisés.

1. Déploiement d'une application Express

• Préparation de l'application pour le déploiement :

Avant de déployer votre application Express, assurez-vous qu'elle est prête pour la production. Voici quelques étapes à suivre :

1. Assurez-vous que toutes les dépendances sont correctement listées dans votre fichier **package.json**.
2. Configurez des variables d'environnement pour stocker les informations sensibles, comme les clés API et les chaînes de connexion aux bases de données.
3. Optimisez les performances en activant la mise en cache, en compressant les fichiers statiques et en utilisant d'autres techniques d'optimisation.
4. Gérez les erreurs de manière appropriée en fournissant des réponses d'erreur génériques aux clients et en enregistrant les erreurs pour le débogage.
5. Testez votre application pour vous assurer qu'elle fonctionne correctement et qu'elle est sécurisée.

• Options de déploiement (Heroku, AWS, etc.) :

Il existe plusieurs options pour déployer votre application Express, notamment :

1. Heroku : Une plateforme cloud qui simplifie le déploiement et la gestion d'applications web. Heroku prend en charge Node.js et offre un processus de déploiement simple et rapide.
2. AWS (Amazon Web Services) : Un ensemble de services cloud qui vous permettent de déployer et de gérer des applications web à grande échelle. Vous pouvez utiliser Elastic Beanstalk ou EC2 pour déployer votre application Express.
3. Google Cloud Platform : Une plateforme cloud proposée par Google qui prend en charge Node.js et permet de déployer et gérer des applications web. Vous pouvez utiliser App Engine ou Cloud Functions pour déployer votre application Express.
4. Microsoft Azure : Une plateforme cloud de Microsoft qui prend également en charge Node.js et offre des services pour déployer et gérer des applications web. Vous pouvez utiliser Azure App Service pour déployer votre application Express.

• Processus de déploiement sur une plateforme spécifique :

Chaque plateforme a son propre processus de déploiement. Voici un exemple de déploiement sur Heroku :

1. Inscrivez-vous et connectez-vous à Heroku (<https://www.heroku.com/>).
2. Installez Heroku CLI (<https://devcenter.heroku.com/articles/heroku-cli>) et connectez-vous avec la commande **heroku login**.
3. Naviguez vers le répertoire de votre application Express et créez un fichier **Procfile** sans extension pour définir le processus à exécuter. Par exemple, ajoutez **web: node app.js** dans le fichier **Procfile**.
4. Initialisez un dépôt Git dans le répertoire de votre application (**git init**), ajoutez tous les fichiers (**git add .**) et validez les modifications (**git commit -m "Initial commit"**).
5. Créez une nouvelle application Heroku avec la commande **heroku create**. Cela ajoutera également un nouveau dépôt distant à votre dépôt Git.
6. Déployez votre application sur Heroku en poussant vos modifications sur le dépôt distant avec la commande **git push heroku master**.
7. Votre application sera maintenant accessible à l'URL générée par Heroku.

Ces étapes vous donneront une idée de base du processus de déploiement sur Heroku. Pour d'autres plateformes, consultez leur documentation spécifique.