Documentation technique

Fichier : 01 - app.js

Tutoriel Node.js avec Express, CORS et dotenv

Ce tutoriel explique comment configurer un serveur Node.js en utilisant le framework Express, en ajoutant une gestion des CORS (Cross-Origin Resource Sharing) et en utilisant dotenv pour gerer les variables d'environnement.

Importation des modules

```
import express from 'express';
import cors from 'cors';
import dotenv from 'dotenv';

import aiRoutes from './routes/ai.js';
import aiServices from './config/ai-services.js';
```

Dans cette section, nous importons les modules necessaires pour notre application.

- `express` est un framework pour Node.js qui simplifie le developpement de serveurs web.
- `cors` est un package Node.js qui fournit un middleware pour activer les CORS avec diverses options.
- `dotenv` est un module qui charge les variables d'environnement a partir d'un fichier `.env` dans `process.env`.
- `aiRoutes` est un module personnalise qui contient les routes pour notre API.
- `aiServices` est un module personnalise qui contient la configuration de nos services d'IA.

Configuration de l'application

```
dotenv.config();

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

app.use(cors());
app.use(express.json());
```

Ici, nous configurons notre application.

- Nous commencons par charger les variables d'environnement a l'aide de `dotenv.config()`.
- Nous instancions ensuite notre application Express et definissons le port sur lequel notre serveur ecoutera.
- Nous ajoutons ensuite le middleware CORS a notre application pour permettre les requetes cross-origin.
- Enfin, nous utilisons `express.json()` pour analyser les corps des requetes entrantes dans un middleware avant vos gestionnaires, disponibles sous la propriete `req.body`.

Definition des routes

```
app.use('/api/ai', aiRoutes);

app.get('/api/ai/services', (req, res) => {
  res.json({ services: aiServices });
});
```

Dans cette section, nous definissons les routes pour notre API.

- Nous utilisons `app.use` pour ajouter le middleware `aiRoutes` a notre application. Toutes les requetes commencant par `/api/ai` utiliseront ce middleware.
- Nous definissons ensuite une route GET pour `/api/ai/services` qui renvoie un objet JSON contenant nos services d'IA.

Demarrage du serveur

```
app.listen(port, () => {
  console.log(`Server listening on http://localhost:${port}`);
});
```

Enfin, nous demarrons notre serveur en ecoutant sur le port specifie. Une fois que le serveur est demarre, un message est affiche dans la console pour indiquer que le serveur est en ecoute.

Fichier : 02 - config_ai-services.js

Tutoriel Node.js : Comprendre le Code d'un Service d'IA

Dans ce tutoriel, nous allons examiner un bloc de code Node.js qui definit un objet contenant des informations sur differents services d'IA. Ce code est un excellent exemple de comment structurer et organiser des informations dans un format facile a comprendre et a utiliser.

Bloc de Code

Voici le bloc de code Node.js en question :

```
const aiServices = {
  llm: [
    { type: 'chatgpt', label: 'OpenAI', purpose: 'Text generation, summarization, Q&A, code
completion' },
    { type: 'claude', label: 'Claude', purpose: 'Structured reasoning, content writing, safe
dialogue' },
    // autres services LLM...
  ],

  tts: [
    { type: 'elevenlabs', label: 'ElevenLabs', purpose: 'High-quality voice synthesis from text,
multilingual' },
    // autres services TTS...
  ],

  // autres categories de services...

};

export default aiServices;
```

Explication du Code

Ce code definit un objet JavaScript `aiServices` qui contient des informations sur differents services d'IA. Chaque cle de l'objet represente une categorie de services d'IA, et la valeur associee est un tableau d'objets, chaque objet representant un service d'IA specifique.

Structure des Services d'IA

Chaque service d'IA est defini par un objet avec les proprietes suivantes :

- `type` : Un identifiant unique pour le service.
- `label` : Le nom du service.
- `purpose` : Une description de ce que le service fait.

Par exemple, `{ type: 'chatgpt', label: 'OpenAI', purpose: 'Text generation, summarization, Q&A, code completion' }` represente un service d'IA de la categorie 'llm' qui est identifie par 'chatgpt', appele 'OpenAI' et qui est utilise pour la generation de texte, la resume, les questions-reponses et la completion de code.

Categories de Services d'IA

Les differentes categories de services d'IA sont :

- `llm` : Language Learning Models, des modeles d'apprentissage automatique pour le traitement du langage naturel.
- `tts` : Text-to-Speech, des services pour convertir le texte en parole.
- `avatar` : Des services pour creer des avatars animes.
- `image` : Des services pour generer des images.
- `agent` : Des services pour creer des agents intelligents.
- `music` : Des services pour generer de la musique.

Chaque categorie contient un tableau de services d'IA specifiques a cette categorie.

Exportation du Module

Enfin, le code utilise `export default aiServices;` pour exporter l'objet `aiServices` afin qu'il puisse etre importe et utilise dans d'autres parties de l'application.

Conclusion

Ce bloc de code Node.js est un bon exemple de comment structurer des informations dans un objet JavaScript. Il montre comment vous pouvez organiser des donnees complexes de maniere logique et facile a comprendre.

Fichier : 03 - mock_llm_chatgpt.mock.js

Tutoriel : Comprendre la fonction `reply` en Node.js

Dans ce tutoriel, nous allons decomposer une fonction Node.js appelee `reply`. Cette fonction est utilisee pour generer une reponse formatee en fonction des parametres recus.

Code

Voici le code de la fonction `reply` :

```
function reply(type, data) {
  const name = (data.name || 'Inconnu').replace('-', ' ');
  const style = data.style || 'neutral';
  const length = data.length || 'medium';
  const llm = data.llm || 'chatgpt';
  const validType = ['biography', 'filmography', 'summary'].includes(type) ? type : 'contenu';

  return `Mock Backend - Demande envoyee a ${llm} pour une ${validType} de "${name}", avec un
style "${style}" et une longueur "${length}".`;
}

export default reply;
```

Explication

Initialisation des variables

La fonction `reply` prend deux parametres : `type` et `data`.

```
const name = (data.name || 'Inconnu').replace('-', ' ');
const style = data.style || 'neutral';
const length = data.length || 'medium';
const llm = data.llm || 'chatgpt';
```

Dans ce bloc de code, nous initialisons quatre variables (`name`, `style`, `length`, `llm`) avec les valeurs correspondantes de l'objet `data`. Si une valeur n'est pas fournie, nous utilisons une valeur par defaut.

Validation du type

```
const validType = ['biography', 'filmography', 'summary'].includes(type) ? type : 'contenu';
```

Ici, nous verifions si le `type` fourni est present dans notre tableau de types valides. Si c'est le cas, nous utilisons le `type` fourni, sinon nous utilisons `'contenu'` comme valeur par defaut.

Construction de la reponse

```
return `Mock Backend - Demande envoyee a ${llm} pour une ${validType} de "${name}", avec un style
```

```
"${style}" et une longueur "${length}".`;
```

Enfin, nous construisons et renvoyons une chaine de caracteres formatee qui contient toutes nos variables.

Conclusion

La fonction `reply` est un exemple simple de comment nous pouvons utiliser les parametres et les valeurs par defaut pour generer une reponse dynamique en Node.js. En comprenant comment cette fonction fonctionne, vous pouvez creer des fonctions similaires qui repondent a vos propres besoins.

Fichier : 04 - mock_llm_claude.mock.js

Tutoriel Node.js : Comprendre la fonction `reply`

Dans ce tutoriel, nous allons decomposer une fonction Node.js appelee `reply`. Cette fonction est concue pour generer une reponse formatee a partir d'un certain type de demande et des donnees associees.

Code de la fonction `reply`

```
function reply(type, data) {
  const name = (data.name || 'Inconnu').replace('-', ' ');
  const style = data.style || 'neutral';
  const length = data.length || 'medium';
  const llm = data.llm || 'claude';
  const validType = ['biography', 'filmography', 'summary'].includes(type) ? type : 'contenu';

  return `Mock Backend - Demande envoyee a ${llm} pour une ${validType} de "${name}", avec un
style "${style}" et une longueur "${length}".`;
}

export default reply;
```

Explication du code

La fonction `reply` prend deux arguments : `type` et `data`. `type` est une chaine de caracteres qui indique le type de contenu demande, et `data` est un objet contenant des informations supplementaires sur la demande.

Definition des constantes

```
const name = (data.name || 'Inconnu').replace('-', ' ');
  const style = data.style || 'neutral';
  const length = data.length || 'medium';
  const llm = data.llm || 'claude';
```

Dans cette section, nous definissons plusieurs constantes a partir des donnees fournies. Si une certaine valeur n'est pas fournie, nous utilisons une valeur par defaut. Par exemple, si `data.name` n'est pas fourni, nous utilisons 'Inconnu' comme valeur par defaut. De plus, si `data.name` contient un tiret, nous le remplacons par un espace.

Validation du type de contenu

```
const validType = ['biography', 'filmography', 'summary'].includes(type) ? type : 'contenu';
```

Ici, nous verifions si le type de contenu demande est valide. Si `type` est 'biography', 'filmography' ou 'summary', nous utilisons cette valeur. Sinon, nous utilisons 'contenu' comme valeur par defaut.

Generation de la reponse

```
return `Mock Backend - Demande envoyee a ${llm} pour une ${validType} de "${name}", avec un style "${style}" et une longueur "${length}".`;
```

Enfin, nous generons la reponse en utilisant une chaine de caracteres formatee. Cette chaine contient toutes les informations que nous avons recueillies et validees.

Conclusion

La fonction `reply` est un exemple de la facon dont nous pouvons traiter et valider des donnees en Node.js. En comprenant comment cette fonction fonctionne, vous pouvez mieux comprendre comment manipuler et valider les donnees dans vos propres applications Node.js.

Fichier : 05 - routes_ai.js

Tutoriel Node.js : Creation d'un routeur Express pour les services LLM

Dans ce tutoriel, nous allons decomposer un bloc de code Node.js qui illustre comment creer un routeur Express pour gerer les requetes vers les services LLM.

Importation des modules necessaires

```
import express from 'express';
import dotenv from 'dotenv';

import chatgptMock from '../mock/llm/chatgpt.mock.js';
import claudeMock from '../mock/llm/claude.mock.js';
import chatgptReal from '../services/llm/chatgpt.service.js';
import claudeReal from '../services/llm/claude.service.js';
```

Dans cette section, nous importons tous les modules necessaires. `express` pour la creation de notre application web, `dotenv` pour la gestion des variables d'environnement. Nous importons egalement des modules de services mock et reels pour deux fournisseurs LLM, `chatgpt` et `claude`.

Configuration et initialisation

```
dotenv.config();

const router = express.Router();
const useMock = process.env.USE_MOCK === 'true';
```

Ici, nous initialisons `dotenv` pour charger les variables d'environnement. Nous creons egalement une nouvelle instance de routeur Express et determinons si nous devons utiliser les services mock ou reels en fonction de la variable d'environnement `USE_MOCK`.

Fonctions auxiliaires

```
function isUnauthorizedError(message) {
  return message.includes('unauthorized') || message.includes('401');
}

function getProvider(llm) {
  const providers = {
    chatgpt: {
      mock: chatgptMock,
      real: chatgptReal,
    },
    claude: {
      mock: claudeMock,
      real: claudeReal,
    },
  };
```

```
  return providers[llm] || null;
}

async function handleLLMRequest(type, llm, data) {
  const provider = getProvider(llm);
  if (!provider) { return { error: 'unknown-provider' }; }

  const fn = useMock ? provider.mock : provider.real;

  return { data: await fn(type, data) };
}
```

Dans cette section, nous definissons trois fonctions auxiliaires. `isUnauthorizedError` verifie si un message d'erreur indique une erreur d'autorisation. `getProvider` retourne le fournisseur LLM approprie en fonction du nom fourni. `handleLLMRequest` gere les requetes LLM en appelant la fonction appropriee du fournisseur LLM et retourne le resultat.

Gestion des requetes

```
router.post('/:type/:llm', async (req, res) => {
  const { type, llm } = req.params;
  const input = req.body;

  try {
    const { data, error } = await handleLLMRequest(type, llm, input);

    if (error) {
      return res.status(400).json({ success: false, llm: llm, data: error });
    }

    return res.json({ success: true, llm: llm, data: data });

  } catch (err) {
    const msg = err.message?.toLowerCase() || '';
    const isUnauthorized = isUnauthorizedError(msg);

    return res.status(500).json({
      success: false,
      llm: llm,
      data: isUnauthorized ? 'unauthorized API KEY' : 'internal-error',
    });
  }
});

export default router;
```

Enfin, nous definissons un gestionnaire pour les requetes POST vers notre routeur. Nous extrayons les parametres de la requete, appelons notre fonction `handleLLMRequest` et renvoyons une reponse appropriee en fonction du resultat. En cas d'erreur, nous verifions si c'est une erreur d'autorisation et renvoyons un message d'erreur approprie.

Et voila ! Vous avez maintenant une meilleure comprehension de la facon dont vous pouvez creer un routeur Express pour gerer les requetes vers les services LLM dans une application Node.js.

Fichier : 06 - services_llm_chatgpt.service.js

Tutoriel : Creation d'une fonction de reponse avec Node.js et l'API OpenAI

Dans ce tutoriel, nous allons examiner une fonction `reply` qui utilise l'API OpenAI pour generer des reponses en fonction de differents styles et longueurs de reponses.

Code complet

Voici le code complet que nous allons decomposer et expliquer :

```
import axios from 'axios';

const styleMap = {
  //...
};

const lengthMap = {
  //...
};

async function reply(type, input) {
  //...
}

export default reply;
```

Importation de la bibliotheque Axios

```
import axios from 'axios';
```

Nous importons la bibliotheque `axios` pour effectuer des requetes HTTP. Axios est une bibliotheque populaire qui offre une API facile a utiliser pour effectuer des requetes HTTP.

Definition des mappages de style et de longueur

```
const styleMap = {
  //...
};

const lengthMap = {
  //...
};
```

Ces deux objets mappent des chaines de caracteres a des descriptions de styles et de longueurs de reponses. `styleMap` definit differents styles de reponses, tandis que `lengthMap` definit differentes longueurs de reponses.

Fonction de reponse

```
async function reply(type, input) {
  //...
}
```

C'est la fonction principale qui genere la reponse. Elle est asynchrone car elle effectue une requete HTTP, qui est une operation asynchrone.

Preparation de la requete

```
const name = input.name || 'inconnu';
const rawStyle = input.style || 'neutral';
const rawLength = input.length || 'medium';

const style = styleMap[rawStyle] || styleMap.neutral;
const length = lengthMap[rawLength] || lengthMap.medium;

const prompt = type === 'summary'
  ? `Fais un resume du film "${name}" avec un style ${style}, ${length}.`
  : `Ecris une biographie de ${name} avec un style ${style}, ${length}.`;
```

Dans cette section, nous preparons les donnees pour la requete. Nous utilisons les valeurs par defaut si certaines donnees ne sont pas fournies.

Envoi de la requete

```
const response = await axios.post(
  'https://api.openai.com/v1/chat/completions',
  {
    model: 'gpt-4-turbo',
    messages: [{ role: 'user', content: prompt }],
  },
  {
    headers: {
      Authorization: `Bearer ${process.env.OPENAI_API_KEY}`,
      'Content-Type': 'application/json',
    },
  },
);
```

Ici, nous envoyons une requete POST a l'API OpenAI. Nous utilisons le modele `gpt-4-turbo` et nous passons notre message dans le corps de la requete.

Gestion des erreurs

```
try {
  //...
} catch (error) {
  //...
}
```

Nous utilisons un bloc `try/catch` pour gerer les erreurs qui peuvent se produire lors de l'envoi de la requete.

Si une erreur se produit, nous la consignons et la renvoyons pour qu'elle puisse etre traitee par le code appelant.

Exportation de la fonction de reponse

```
export default reply;
```

Enfin, nous exportons la fonction `reply` pour qu'elle puisse etre utilisee dans d'autres modules.

Fichier : 07 - services_llm_claude.service.js

Tutoriel : Utilisation de l'API Claude avec Node.js

Ce tutoriel vous guide a travers un exemple de code Node.js qui utilise l'API Claude pour generer des reponses stylisees et de longueur variable.

Code complet

Voici le code complet que nous allons decomposer et expliquer dans ce tutoriel.

```
import axios from 'axios';

const styleMap = {
  //...
};

const lengthMap = {
  //...
};

async function reply(type, input) {
  //...
}

export default reply;
```

Importation de la bibliotheque Axios

```
import axios from 'axios';
```

Nous commencons par importer la bibliotheque Axios, qui est une bibliotheque JavaScript promise-based utilisee pour faire des requetes HTTP.

Cartes de style et de longueur

```
const styleMap = {
  //...
};

const lengthMap = {
  //...
};
```

Ces deux objets sont des cartes de style et de longueur. Ils associent des identifiants de style et de longueur a des descriptions textuelles. Ces descriptions seront utilisees pour construire l'invite que nous allons envoyer a l'API Claude.

Fonction de reponse

```
async function reply(type, input) {
  //...
}
```

C'est ici que la magie opere. La fonction `reply` est une fonction asynchrone qui prend deux arguments : `type` et `input`. `type` peut etre soit 'summary' soit autre chose, et `input` est un objet qui doit contenir les cles `name`, `style` et `length`.

Construction de l'invite

```
const name = input.name || 'inconnu';
const rawStyle = input.style || 'neutral';
const rawLength = input.length || 'medium';

const style = styleMap[rawStyle] || styleMap.neutral;
const length = lengthMap[rawLength] || lengthMap.medium;

const prompt = type === 'summary'
  ? `Fais un resume du film "${name}" avec un style ${style}, ${length}.`
  : `Ecris une biographie de ${name} avec un style ${style}, ${length}.`;
```

Dans cette section, nous construisons l'invite a envoyer a l'API Claude. Nous utilisons les valeurs de `name`, `style` et `length` fournies dans l'objet `input`, ou des valeurs par defaut si elles ne sont pas fournies.

Envoi de la requete a l'API Claude

```
const response = await axios.post(
  'https://api.anthropic.com/v1/messages',
  {
    model: 'claude-3-5-sonnet-20240620',
    max_tokens: 1000,
    messages: [{ role: 'user', content: prompt }],
  },
  {
    headers: {
      'x-api-key': process.env.ANTHROPIC_API_KEY,
      'anthropic-version': '2023-06-01',
      'Content-Type': 'application/json',
    },
  },
);
```

Nous utilisons Axios pour envoyer une requete POST a l'API Claude. Nous passons l'invite que nous avons construite dans la section precedente dans le corps de la requete.

Gestion des erreurs

```
const result = response.data.content?.[0]?.text;
if (!result) { throw new Error('Reponse vide de Claude.'); }

return result;
```

Enfin, nous verifions si nous avons recu une reponse de l'API Claude. Si ce n'est pas le cas, nous lancons une erreur. Si nous avons recu une reponse, nous la renvoyons.

Exportation de la fonction de reponse

```
export default reply;
```

Nous exportons la fonction `reply` pour qu'elle puisse etre utilisee dans d'autres parties de notre application.