Documentation technique

Fichier : 01 - app.js

Tutoriel Node.js avec Express, CORS et Dotenv

Dans ce tutoriel, nous allons passer en revue un exemple de code Node.js utilisant Express, CORS et Dotenv. Nous allons expliquer chaque bloc de code en detail pour vous aider a comprendre comment ils fonctionnent ensemble.

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. Cela comprend `express` pour la gestion du serveur, `cors` pour gerer les requetes cross-origin, `dotenv` pour charger les variables d'environnement, ainsi que `aiRoutes` et `aiServices` qui sont des modules specifiques a notre application.

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 utilisons `dotenv.config()` pour charger les variables d'environnement. Ensuite, nous creons une nouvelle instance d'Express et definissons le port sur lequel notre serveur sera a l'ecoute. Enfin, nous utilisons `app.use(cors())` pour permettre les requetes cross-origin et `app.use(express.json())` pour analyser les requetes entrantes avec des charges utiles JSON.

Definition des routes

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

Dans cette section, nous definissons les routes pour notre application. Nous utilisons `app.use('/api/ai', aiRoutes)` pour dire a Express que toutes les requetes commencant par '/api/ai' doivent etre gerees par `aiRoutes`.

Route pour obtenir les services AI

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

Ici, nous definissons une route specifique pour obtenir la liste des services AI. Cette route repond a une requete GET en renvoyant un objet JSON contenant la liste des services AI.

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 pret, un message est affiche dans la console pour indiquer que le serveur est en ecoute.

Et voila ! Vous avez maintenant une meilleure comprehension de ce code Node.js. N'hesitez pas a l'experimenter et a l'adapter a vos propres besoins.

Fichier : 02 - config_ai-services.js

Tutoriel : Comprendre le code Node.js pour les services d'IA

Ce tutoriel va vous aider a comprendre le code Node.js utilise pour definir une collection de services d'IA. Nous allons examiner le code, puis expliquer les concepts cles et les roles des differentes sections du code.

Code

```
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' },
    { type: 'gemini', label: 'Gemini', purpose: 'Multimodal LLM for text and image
understanding' },
    { type: 'mistral', label: 'Mistral', purpose: 'Open-source LLM for high-performance text/code
tasks' },
    { type: 'perplexity', label: 'Perplexity AI', purpose: 'Web-augmented search engine powered
by LLM' },
    { type: 'deepseek', label: 'DeepSeek', purpose: 'Code generation, explanation and debugging
assistant' },
  ],

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

  avatar: [
    { type: 'did', label: 'D-ID', purpose: 'Animate a still photo with audio or text' },
    { type: 'heygen', label: 'Heygen', purpose: 'Generate talking avatar videos from script' },
    { type: 'jogg', label: 'Jogg AI', purpose: 'Create realistic talking avatars from custom
photos' },
  ],

  image: [
    { type: 'leonardo', label: 'Leonardo AI', purpose: 'Create illustrations, concept art and
product visuals' },
    { type: 'midjourney', label: 'MidJourney', purpose: 'Stylized artistic image generation from
prompt' },
    { type: 'kling', label: 'Kling AI', purpose: 'Future video generation from text (Sora-level
quality)' },
  ],

  agent: [
    { type: 'langchain', label: 'LangChain', purpose: 'Chain tools, memory, and LLMs into
intelligent agents' },
    { type: 'llamaindex', label: 'LlamaIndex', purpose: 'Connect LLMs to data sources, documents,
```

```
and files' },
  ],

  music: [
    { type: 'suno', label: 'Suno AI', purpose: 'Generate full songs with lyrics, melody, and
vocals' },
    { type: 'udio', label: 'Udio AI', purpose: 'Generate high-quality vocal music tracks from
prompt' },
  ],
};

export default aiServices;
```

Explication du code

Structure de base

Le code definit une constante `aiServices` qui est un objet JavaScript. Cet objet contient plusieurs proprietes qui correspondent a differentes categories de services d'IA : `llm`, `tts`, `avatar`, `image`, `agent` et `music`. Chaque categorie contient un tableau d'objets, ou chaque objet represente un service d'IA specifique.

Description des services d'IA

Chaque service d'IA est defini par un objet avec trois proprietes : `type`, `label` et `purpose`.

- `type` : Il s'agit d'une chaine de caracteres unique qui identifie le service d'IA.
- `label` : Il s'agit d'un nom plus descriptif ou d'une etiquette pour le service d'IA.
- `purpose` : Il s'agit d'une description de ce que le service d'IA fait ou de son utilisation prevue.

Exportation du module

A la fin du code, nous exportons l'objet `aiServices` en tant que module par defaut. Cela signifie que lorsque ce fichier est importe dans un autre fichier, l'objet `aiServices` sera directement accessible.

Fichier : 03 - mock_llm_chatgpt.mock.js

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

Dans ce tutoriel, nous allons decomposer et expliquer une fonction Node.js appelee `reply`. Cette fonction est utilisee pour generer une reponse formatee basee sur les parametres d'entree.

Code de la fonction

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 du code

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

Gestion des valeurs par defaut

La premiere partie de la fonction est dediee a la gestion des valeurs par defaut. Si certaines proprietes dans l'objet `data` ne sont pas fournies, la fonction assigne des valeurs par defaut a ces proprietes.

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

Validation du type de contenu

La fonction verifie ensuite si le `type` fourni est valide. Si ce n'est pas le cas, elle assigne 'contenu' comme valeur par defaut.

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

Generation de la reponse

Enfin, la fonction retourne une chaine de caracteres formatee qui resume la demande de l'utilisateur.

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

Exportation de la fonction

La fonction est ensuite exportee pour etre utilisee dans d'autres parties du code.

```
export default reply;
```

Conclusion

En resume, la fonction `reply` est une fonction Node.js qui genere une reponse formatee basee sur les parametres d'entree. Elle gere egalement les valeurs par defaut et valide le type de contenu demande.

Fichier : 04 - mock_llm_claude.mock.js

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

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

Code source

Voici le code source 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;
```

Explications

Parametres de la fonction

La fonction `reply` prend deux parametres : `type` et `data`. `type` est une chaine de caracteres qui represente le type de demande, tandis que `data` est un objet contenant plusieurs proprietes qui definissent les details de la demande.

Definition des variables

Dans la fonction, nous definissons plusieurs constantes (`name`, `style`, `length`, `llm`, `validType`) en utilisant les valeurs de l'objet `data` ou des valeurs par defaut si les proprietes correspondantes de `data` sont `undefined`.

Validation du type

La constante `validType` est definie en verifiant si `type` est inclus dans un tableau de types valides. Si c'est le cas, `validType` est egal a `type`, sinon il est defini a `'contenu'` par defaut.

Generation de la reponse

Enfin, la fonction retourne une chaine de caracteres formatee qui inclut toutes les constantes definies precedemment. Cette chaine represente la reponse generee par la fonction.

Conclusion

La fonction `reply` est un exemple simple de comment une fonction peut prendre des parametres, les valider, les utiliser pour definir des variables et finalement generer une reponse. En comprenant comment elle fonctionne, vous pouvez creer vos propres fonctions pour generer des reponses ou des messages formates en fonction de differents parametres d'entree.

Fichier : 05 - routes_ai.js

Tutoriel Node.js : Gestion des requetes avec Express et Dotenv

Dans ce tutoriel, nous allons decomposer un bloc de code Node.js qui utilise les modules `express` et `dotenv`, ainsi que des services et des mocks specifiques pour gerer les requetes.

Code complet

```
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';

dotenv.config();

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

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) };
}
```

```
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;
```

Explications

Importation des modules et configuration

```
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';

dotenv.config();

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

Dans cette section, nous importons les modules necessaires pour notre application. Nous utilisons `express` pour gerer notre serveur HTTP et `dotenv` pour gerer les variables d'environnement. Nous importons egalement des mocks et des services reels pour deux fournisseurs : `chatgpt` et `claude`.

Nous configurons ensuite `dotenv` pour qu'il puisse lire les variables d'environnement de notre fichier `.env`. Ensuite, nous initialisons un routeur `express` et determinons si nous devons utiliser des mocks ou des services reels en fonction de la variable d'environnement `USE_MOCK`.

Fonctions utilitaires

```
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;
}
```

Ici, nous definissons deux fonctions utilitaires. `isUnauthorizedError` verifie si un message d'erreur contient des indications d'une erreur d'autorisation. `getProvider` recupere le bon fournisseur (mock ou reel) en fonction du parametre `llm`.

Gestion des requetes

```
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) };
}

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',
    });
  }
});
```

Dans cette section, nous definissons une fonction `handleLLMRequest` pour gerer les requetes a nos fournisseurs. Elle recupere le bon fournisseur, execute la fonction appropriee (mock ou reelle) et renvoie les donnees.

Ensuite, nous definissons une route `POST` pour notre routeur express. Cette route extrait les parametres de la requete, les passe a `handleLLMRequest`, et renvoie une reponse appropriee en fonction du succes ou de l'echec de la requete. En cas d'erreur, elle verifie si l'erreur est due a une autorisation non valide et renvoie un message d'erreur approprie.

Fichier : 06 - services_llm_chatgpt.service.js

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

Dans ce tutoriel, nous allons voir comment utiliser l'API OpenAI pour generer des reponses automatiques en fonction d'un style et d'une longueur definis. Nous utiliserons Node.js et la bibliotheque axios pour effectuer les requetes HTTP.

Code complet

Voici le code complet que nous allons detailler :

```
import axios from 'axios';

const styleMap = {
  //...
};

const lengthMap = {
  //...
};

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

export default reply;
```

Details du code

Importation des dependances

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

Nous utilisons la bibliotheque `axios` pour effectuer des requetes HTTP. Elle nous permettra de communiquer avec l'API OpenAI.

Definition des styles et longueurs

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

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

Ces deux objets definissent les differents styles et longueurs de reponses que notre fonction peut generer. Ils sont utilises pour construire l'invite que nous enverrons a l'API OpenAI.

Fonction de reponse

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

C'est la fonction principale de notre code. Elle prend en entree un type (qui peut etre 'summary' ou autre chose) et un objet `input` qui doit contenir un `name`, un `style` et une `length`. Elle utilise ces informations pour construire une invite et obtenir une reponse de l'API OpenAI.

Gestion des erreurs

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

Cette partie du code gere les erreurs qui peuvent survenir lors de la communication avec l'API OpenAI. Elle affiche un message d'erreur approprie en fonction du code d'erreur recu.

Exportation de la fonction

```
export default reply;
```

Finalement, nous exportons notre fonction `reply` pour pouvoir l'utiliser dans d'autres parties de notre application.

Conclusion

Ce code vous permet d'interagir avec l'API OpenAI et de generer des reponses automatiques en fonction de differents styles et longueurs. Vous pouvez l'adapter a vos besoins en ajoutant de nouveaux styles et longueurs, ou en modifiant la fonction `reply` pour qu'elle accepte d'autres types d'entrees.

Fichier : 07 - services_llm_claude.service.js

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

Dans ce tutoriel, nous allons decomposer un bloc de code Node.js qui utilise le package `axios` pour faire des requetes HTTP a une API. Le but de ce code est de generer une reponse basee sur differents parametres d'entree.

Code complet

```
import axios from 'axios';

const styleMap = {
  //...
};

const lengthMap = {
  //...
};

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

export default reply;
```

Importation du package Axios

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

Nous commencons par importer le package `axios`, qui est une bibliotheque JavaScript promise-based utilisee pour faire des requetes HTTP.

Definition des cartes de style et de longueur

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

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

Ensuite, nous definissons deux objets : `styleMap` et `lengthMap`. Ces objets sont utilises pour mapper les entrees brutes de style et de longueur a leurs descriptions respectives.

Creation de la fonction de reponse

```
async function reply(type, input) {
```

```
  //...
}
```

La fonction `reply` est definie comme une fonction asynchrone, ce qui signifie qu'elle retourne une promesse. Elle prend deux parametres : `type` et `input`. `type` determine le type de reponse a generer (par exemple, un resume ou une biographie) et `input` est un objet contenant des informations supplementaires necessaires pour generer la reponse.

Gestion des erreurs

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

La fonction `reply` utilise un bloc `try...catch` pour gerer les erreurs qui peuvent survenir lors de l'execution du code. Si une erreur se produit, elle est capturee et traitee dans le bloc `catch`.

Envoi de la requete HTTP

```
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',
    },
  },
);
```

En utilisant `axios.post`, nous envoyons une requete HTTP POST a l'API. Les parametres de cette methode comprennent l'URL de l'API, le corps de la requete et les en-tetes de la requete.

Exportation de la fonction de reponse

```
export default reply;
```

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