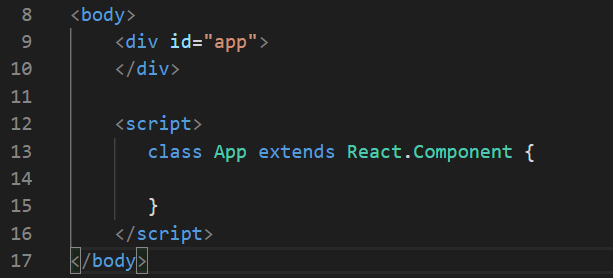
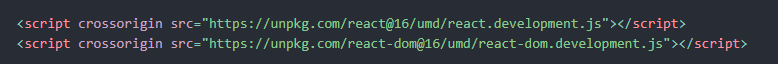
[**https://github.com/m2i-training/react101**](https://github.com/m2i-training/react101)

**Exercice 1: Création d'application React via CDN, JSX, ES6 et Babel**

Créer une classe nommée App qui hérite de React.component:



Pour pouvoir l’utiliser, React.Component devrait être importé depuis l[es liens CDN disponibles sur le site officiel de React](https://reactjs.org/docs/cdn-links.html):



Par la suite, nous allons utiliser le **react-dom** pour afficher le contenu notre div ayant l'id "*app*". Pour cela, notre classe *App* aurait besoin d'une méthode, cette méthode s'appelle **render()**.

La méthode ***render()*** est la seule méthode obligatoire dans un composant de type class dans React.

La méthode render contient un return dans lequel on va écrire du JSX (Javascript XML).

JSX étant toujours recommandé, on pourrait toujours utiliser du javascript dans le ***return()***.

Pour la fonction render, on a besoin d'un élément parent. Ca va être le composant que nous allons exporter:

- C'est un div à l'intérieur de return => JSX l'accept.

- Si on ajoute un autre div, JSX ne l'accepte pas (JSX a besoin d'un seul élément parent).

Pour ajouter une classe CSS, on peut pas utiliser class car ça crée de l'ambiguïté avec le mot clé class dans ES6. Pour cela, on utilise ***className***: **<div className=""></div>**

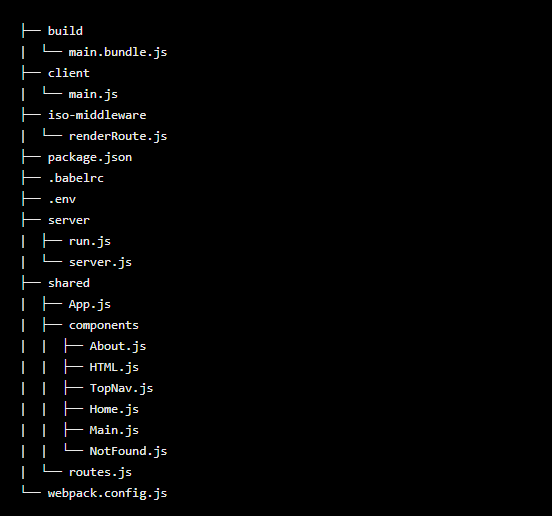


**Exercice 2: Configuration d'une application Web isomorphe**

**1- Structure du projet**

**Le projet est divisé en:**

* Code spécifique au serveur,
* Code spécifique au client pour le rendu de la SPA,
* Code partagé, qui prend en charge à la fois le rendu serveur et client.



**1- Rendu Serveur**

**1.** Webpack.config.js crée /build/main.bundle.js lorsque l'application est créée pour la première fois.

**2.** Main.bundle.js est utilisé dans HTML.js

**3.** HTML.js est un composant React utilisé pour générer le modèle HTML que le serveur envoie au navigateur lorsqu'il reçoit une GET demande HTTP du navigateur (appelée la charge initiale).

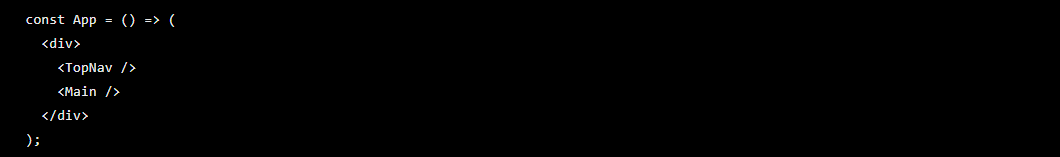
**4.** renderRoute.js rend une version statique de App(en utilisant le StaticRouter comme conteneur) dans le modèle HTML dans HTML.js, convertit tout en chaîne à l'aide de la prise en charge du rendu du serveur React, puis envoie la version finale de la chaîne du HTML à envoyer pour la requête HTTP GET.

**5.** Server.js répond à la demande de chargement initiale en obtenant renderRoutes de répondre avec un code HTML.

**2- Rendu Client**

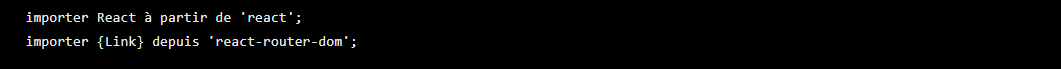
**3- Code partagé**

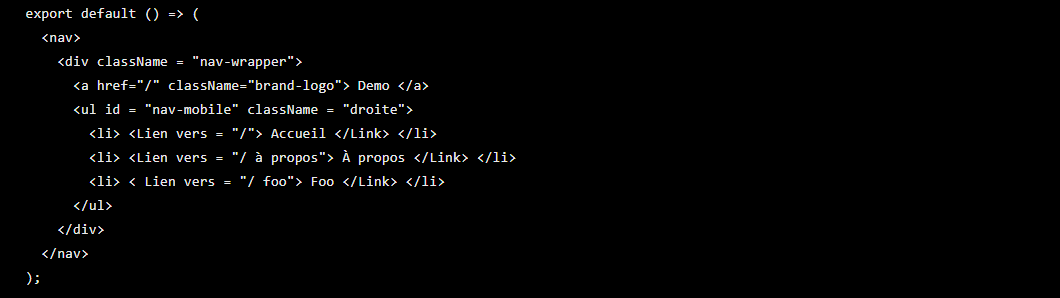
Le principal point d'entrée du code partagé est le composant **<App>**:



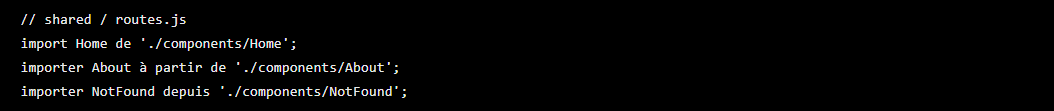


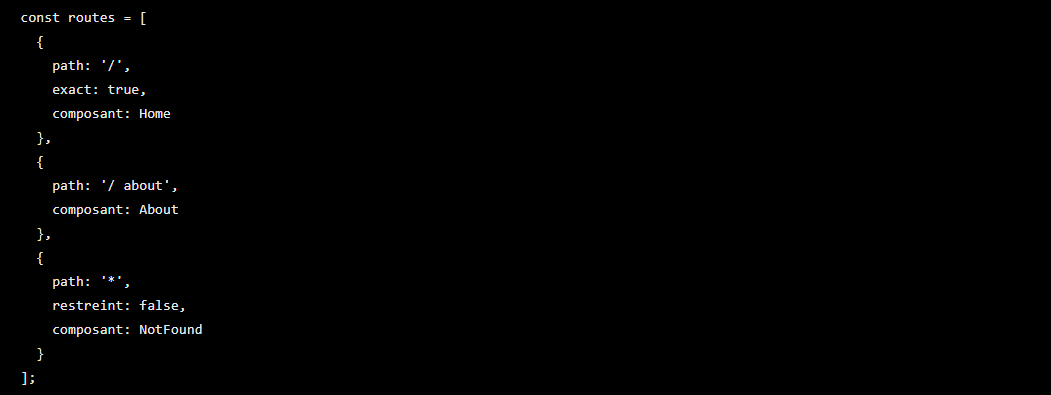
C'est un composant React supérieur assez standard, qui utilise des sous-composants pour rendre différentes pages. **<TopNav>** définit la navigation dans l'application à l'aide du **<Link>** composant React Router:





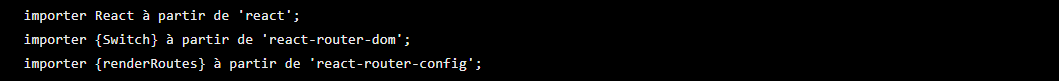
Le mappage pour la page à servir en fonction de l'itinéraire est contenu dans routes.js, qui est importé dans le <Main>composant.



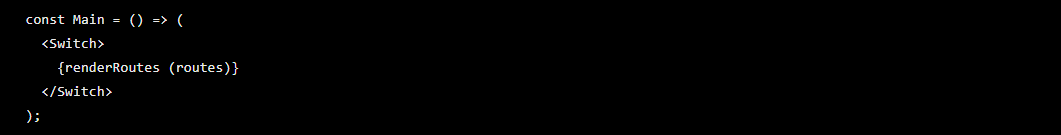




Dans le <Main>composant, la fonction *react-router-config* **renderRoutes** est utilisée pour générer le <Route>composant en fonction du chemin d'accès au mappage de composant défini dans routes.

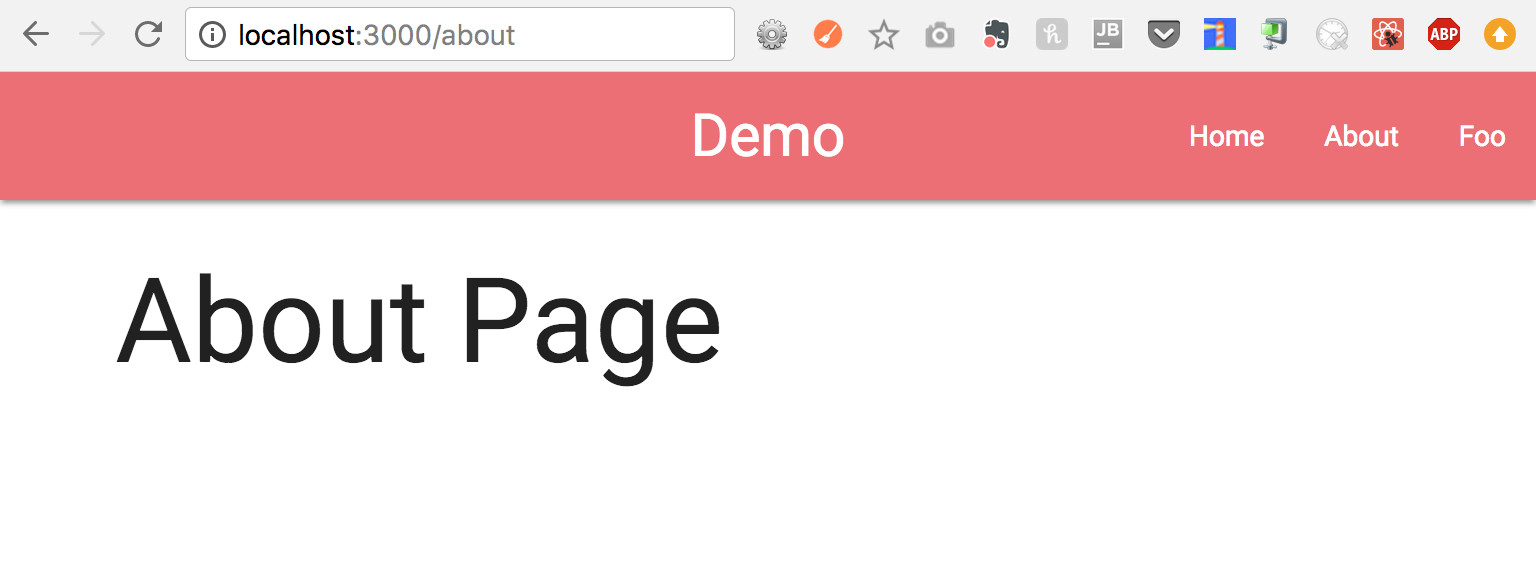
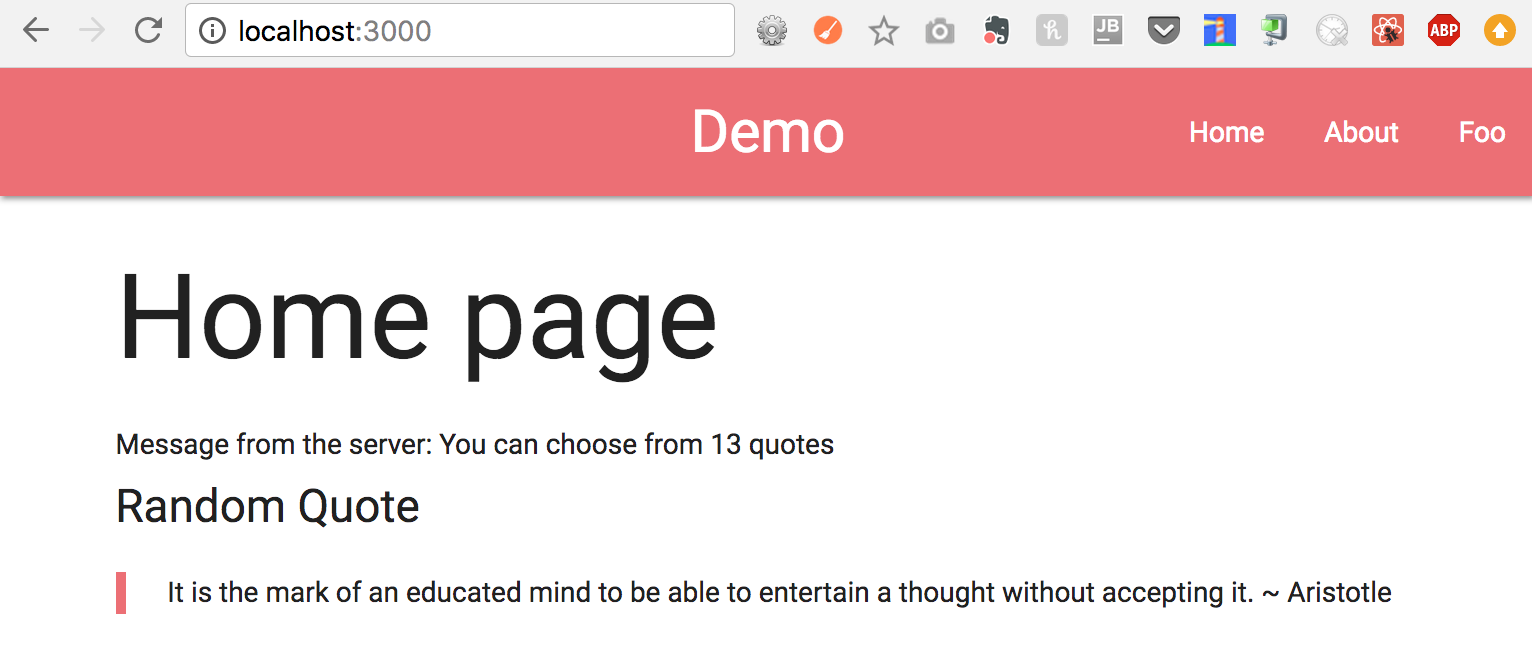


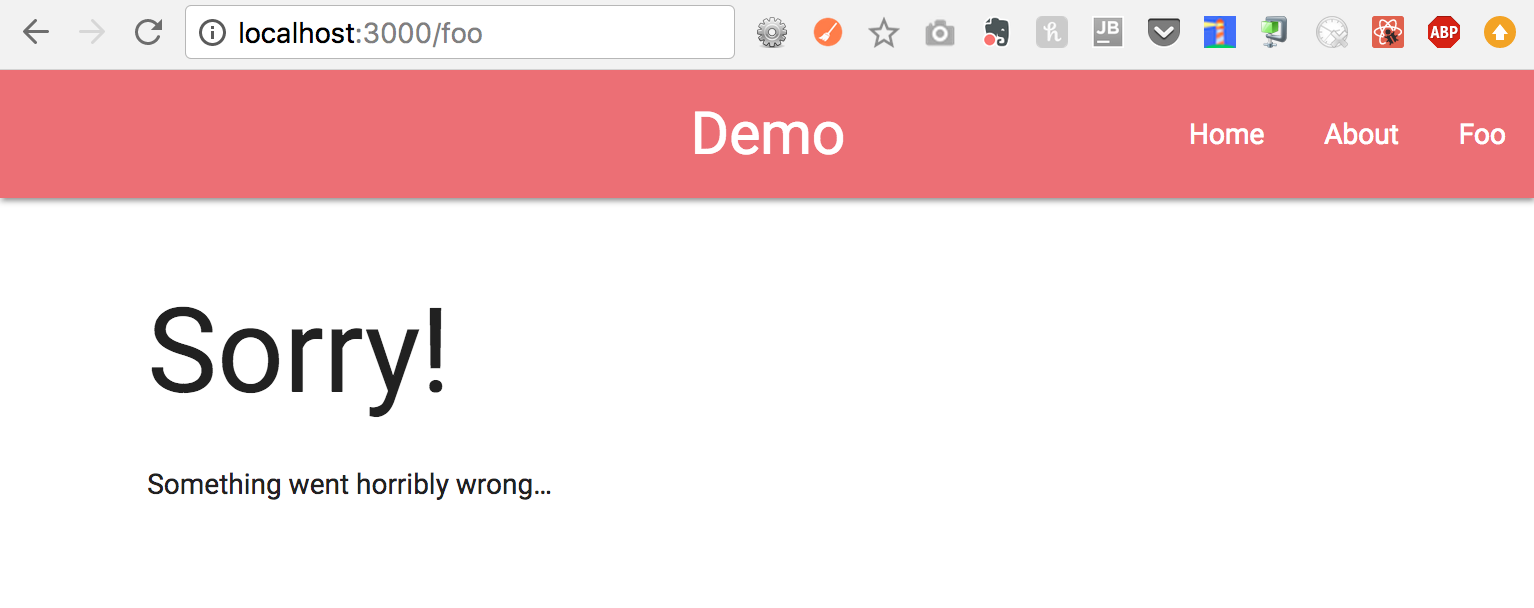






Comme défini dans routes, les pages que vous pouvez rendre sont Accueil, À propos et Non trouvées, comme indiqué ci-dessous:



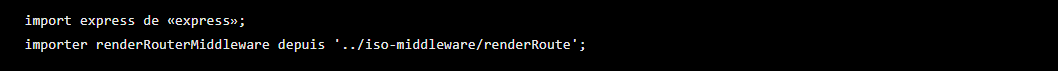


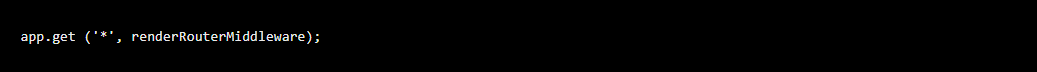
#### **Rendu du serveur**

L'idée est que nous voulons que le serveur envoie le code HTML de la page d'accueil lorsque nous tapons localhost:3000 dans le navigateur et appuyez sur Entrée.

Nous commençons par monter la fonction middleware pour rendre l'application sur l'application serveur comme indiqué server.js ci-dessous:







**renderRouterMiddleware** contient toute la logique de création de la chaîne HTML à l'aide des composants du dossier shared.

**renderRouterMiddleware** est l'un des fichiers les plus importants de notre projet, car il a la logique de rendre l'application isomorphe.

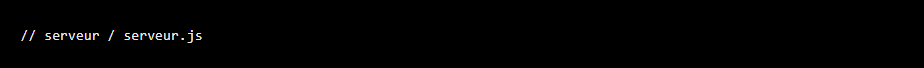
Pour la plupart, la partie de rendu côté serveur du code est assez passe-partout, mais a les ingrédients secrets pour le rendu côté serveur d'un HTML qui permet à l'application cliente de prendre le relais après la charge initiale. Plus précisément, le composant <HTML> importé ici pour le rendu côté serveur contient le lien avec l'application cliente.

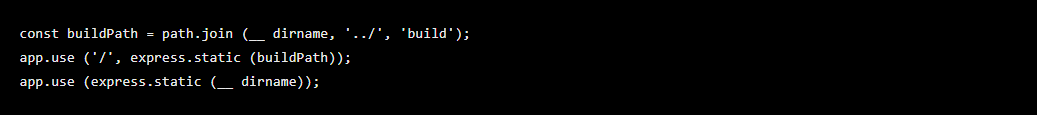
Mais avant de vous montrer le code <HTML>, nous devons passer en revue quelques autres choses.

**Note:** Une autre chose à noter est que pour le rendu du serveur, nous voulons encapsuler notre composant <App> dans React Router <StaticRouter> avant de tout convertir en utilisant la ***renderToStringfonction*** de React . Pour le rendu client, dont nous allons discuter ensuite, nous voulons utiliser <BrowserRouter>.

#### **Rendu client**

Le code serveur ci-dessous fournit au navigateur tout le code dont le navigateur a besoin pour rendre un SPA:





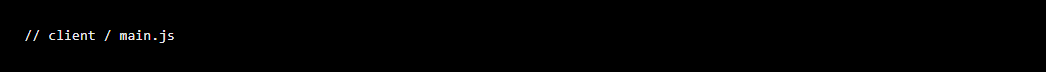
Ce bloc de code indique au serveur de servir les actifs statiques du dossier de construction à localhost:3000/.

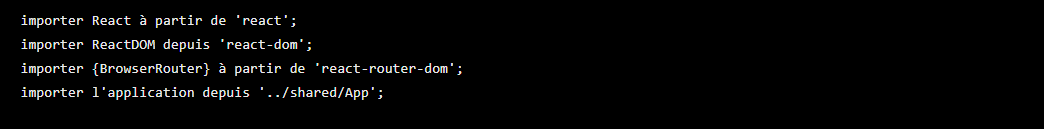
Comme indiqué dans la structure de fichiers ci-dessus, il n'y a qu'un seul fichier -  main.bundle.js  - dans le dossier de génération. Si vous tapez localhost:3000/main.bundle.js dans votre navigateur, vous verrez un tas de JavaScript, qui contient du code de notre dossier shared qui a été transposé depuis ES6 vers une version antérieure de JavaScript.

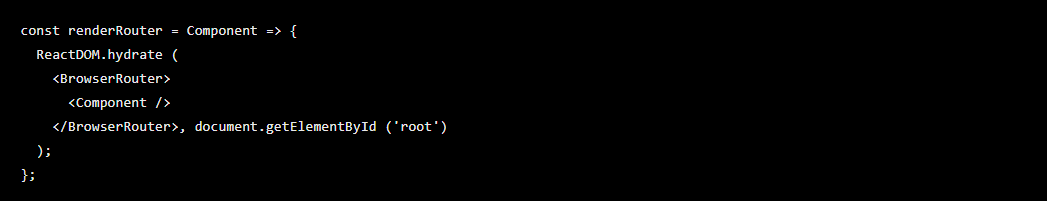
Main.bundle.js est créé par Webpack. Dans [**package**](https://github.com/xiaoyunyang/isomorphic-router-demo/blob/master/package.json)**.**[**json**](https://github.com/xiaoyunyang/isomorphic-router-demo/blob/master/package.json), des scripts ont été configurés pour exécuter une build avant de démarrer le serveur, donc le main.bundle.js est reconstruit à chaque démarrage du serveur.

La définition de build est dans notre webpack.config.js fichier, qui définit ***./client/main.js*** comme l'entrée pour la build.

***Main.js*** et tout ce qu'il utilise est regroupé **main.bundle.js.** Voici le code pour main.js:









La première chose que vous avez probablement remarquée est qu'elle **ReactDOM.hydrate** est utilisée à la place de **ReactDOM.render**. C'est parce que nous voulons attacher l'application rendue par le client à la **root** div du HTML rendu par le serveur .

Rappelez-vous que j'ai dit plus tôt que le composant **<HTML>** est le lien entre le HTML rendu par le serveur et l'application rendue par le client?

**<HTML>**, qui est utilisé par le serveur pour rendre le code HTML pour la charge initiale, crée un div **root** et se charge dynamiquement **main.bundle.js** dans une balise de script. **C'est ce qui fait que cette application isomorphe fonctionne!**

#### **Application isomorphe en action**

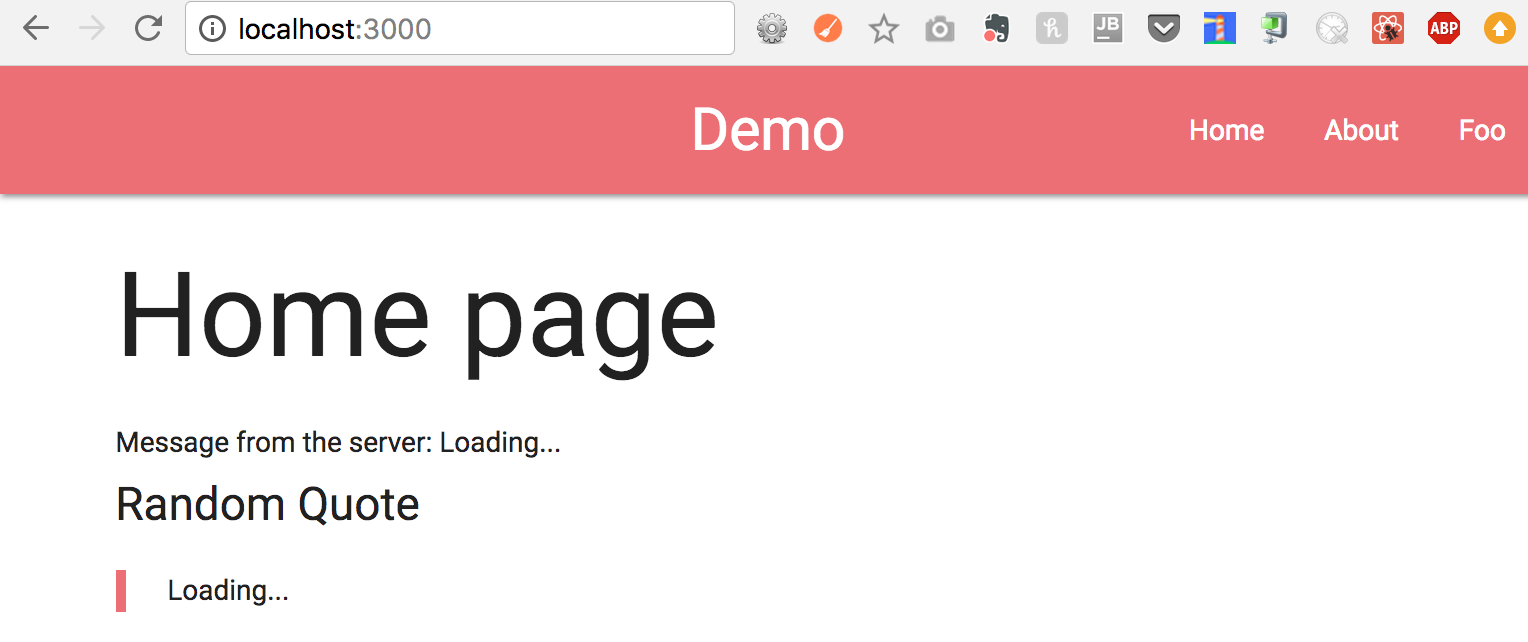
Après avoir démarré l'application à l'aide de npm start, tapez **localhost:3000** dans la barre d'adresse de votre navigateur, appuyez sur Entrée, et vous verrez la page d'accueil s'afficher après que la roue de chargement de la page tourne un peu dans l'onglet du navigateur.

La rotation de la roue de chargement de page indique que le serveur a fait un travail pour vous fournir cette page. Si vous avez cliqué sur About et Foo dans la barre de navigation, vous verrez la page *About* et la page *NotFound* se charger sans que la roue de chargement de page ne tourne dans l'onglet du navigateur.

Cela vous indique que le mode SPA a été activé et gère la navigation de la page en fonction des événements de clic. En fait, l'application peut même s'exécuter lorsque vous arrêtez le serveur.

Allez-y et arrêtez le serveur depuis le terminal pour voir que vous pouvez toujours cliquer pour charger les pages comme avant… mais avec une différence:

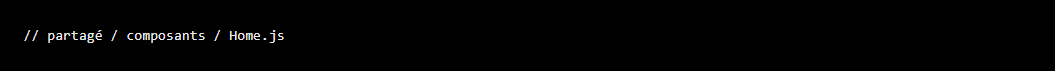
Au lieu d'un message du serveur et d'une citation aléatoire, vous voyez le mot «Chargement».

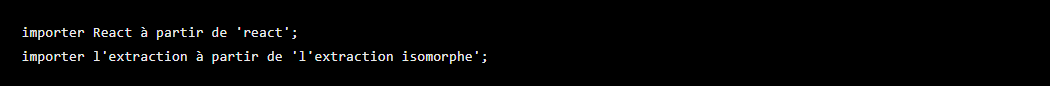
****

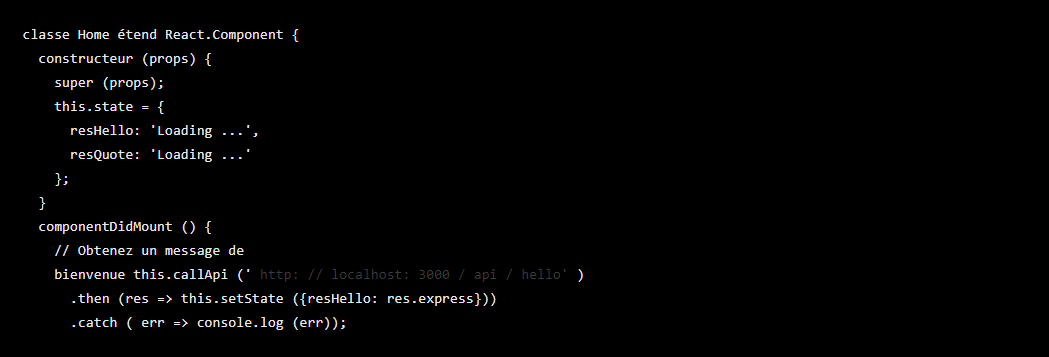
C'est par conception. On a essayé de rendre l'application plus intéressante en lui faisant livrer un devis inspirant aléatoire sur la page d'accueil chaque fois que vous accédez à la page d'accueil via l'application cliente **TopNav** ou en la chargeant directement depuis le serveur.

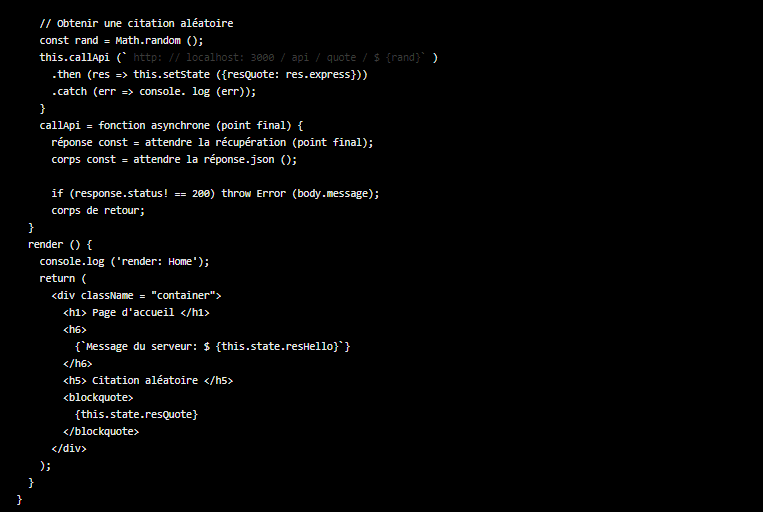
Il s'agit également de démontrer un modèle de conception courant dans les applications Web modernes selon lequel des parties du contenu de la page sont chargées après le chargement de la page via une extraction asynchrone à partir d'une API.

Le composant <Home> récupère certaines données de deux points de terminaison API juste après leur montage.

****

****

****

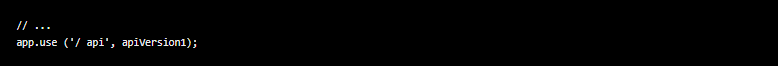
****

****

Le code du serveur responsable de la livraison des données à ces points de terminaison API est affiché ici:

****

****

****

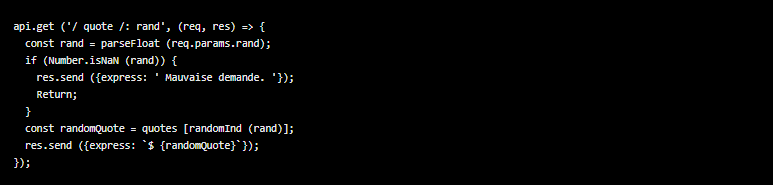
et ici:

****

****

****

****

****

****

Pour plus de simplicité, On a codé en dur le JSON quotes directement dans le code api mais vous pouvez avoir les citations dans un fichier quotes.json ou stockées dans une base de données et utiliser un middleware express pour les récupérer avant utilisation.