Gatsby

Gatsby, ou GatsbyJS, est un générateur de site statique (SSG) construit sur Node.js en utilisant ReactJS et GraphQL. Il s'agit d'un projet openSource né en en mai 2015 auquel ont contribué plus de 1350 développeurs.

Gatsby récupère des données à partir de diverses sources, notamment des sites Web existants, des appels API et des fichiers via GraphQL, et construit le site statique en fonction des réglages de configuration spécifiés.

Introduction à Gatsby-JS

Afin de débuter le projet dans de bonnes conditions, il a fallu me renseigner au préalable sur Gatsby-JS, ne l’ayant jamais utilisé.

Pour cela, le site de Gatsby propose un tutoriel d’introduction. (https://www.gatsbyjs.com/docs/tutorial/)

Cet exercice nous en apprendre plus sur le fonctionnement de Gatsby-JS. Il permet le développement d’un site de blog qui prend en charge des images (statiques et dynamiques) et des fichiers MDX, grâce à des plugins.

Le site créait dynamiquement de nouvelles pages à partir de nœuds de couche de données, en nommant les fichiers avec une syntaxe précise.

Projet

Mon stage consistait à apprendre un nouveau langage informatique : Gatsby JS, dans le but de le comparer à d'autres technologies JavaScript.

Le site web final doit permettre de faire une recherche parmi les données d'une API GraphQL, puis de retourner son résultat.

L'API utilisée est l'API Rick and Morty, utilisable gratuitement. (*https://rickandmortyapi.com/*)

Le projet doit donc comporter plusieurs éléments: une liste de toutes les données de l'API (les personnages), une fiche détaillée pour chaque personnage, ainsi qu'une partie recherche permettant de chercher le personnage souhaité et d'alors obtenir ses informations.

La réalisation du projet se divise en plusieurs étapes, présentées ci dessous.

Le code du projet est présent sur mon répertoire Github : *https://github.com/lpopczyk/gatsby-rickandmorty*

Le fichier .zip :

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/b8720421-3d76-451f-81db-facd15d3753d/gatsby_api.zip

Étape 1 : Configuration de l'environnement de développement

Pour commencer, Gatsby nécessite l'installation de plusieurs outils : Node.js, Git, l'Interface de ligne de commande Gatsby (Gatsby CLI), puis Visual Studio Code. Il est recommandé d'utiliser Hombrew pour l'installation sur Mac, via le Terminal.

Node : `brew install node`

Node.js est un environnement qui exécute du code JavaScript en dehors d'un navigateur Web. Gatsby est construit avec Node.js. Pour être opérationnel avec Gatsby, vous devez avoir la version 14.15 (ou plus récente) de Node.js.

Git : `brew install git`

Git est un système de contrôle de version distribué gratuit et open source. C'est un outil qui vous aide à enregistrer différentes versions de votre code. Cela aide également les coéquipiers à travailler ensemble sur la même base de code en même temps.

Gatsby CLI : `npm install -g gatsby-cli`

L'interface de ligne de commande (CLI) Gatsby est un outil qui permet de créer rapidement de nouveaux sites alimentés par Gatsby et d'exécuter des commandes pour développer des sites Gatsby.

La CLI est un package npm publié, ce qui signifie que vous pouvez l'installer à l'aide de npm.

Visual Studio Code : (https://code.visualstudio.com/#alt-downloads)

Visual Studio Code (également appelé VS Code, en abrégé) est un éditeur de code populaire que l'on peut utiliser pour écrire du code pour un projet.

sources : https://www.gatsbyjs.com/docs/tutorial/part-0/

Étape 2 : Création de l'application

Une fois l'ordinateur correctement configuré, il faut maintenant créer le projet.

Pour cela, il faut se placer dans le répertoire dans lequel on souhaite créer son projet (ici `cd Desktop`) puis taper : `gatsby new`

Puis, il faut entrer les différentes informations demandées par l'invité de commande.

Soit : le nom du site Gatsby, le nom du répertoire, le type utilisé, le choix d'un CMS, le choix d'un système de style, le choix d'installer d'autres plugins.

Une fois le code du site généré, il est maintenant possible de l'exécuter localement, en démarrant le serveur de développement local.

Pour se faire, il faut se placer dans le répertoire du site (ici `cd gatsby_api`) puis entrer : `gatsby develop`.

Après quelques instants, la ligne de commande affiche un message indiquant que le serveur de développement est prêt à fonctionner et est disponible à l'adresse : http://localhost:8000/

sources : https://www.gatsbyjs.com/docs/tutorial/part-1/

Étape 3 : Structuration du projet

Pour créer la structure de page de base de notre site, il est nécessaire de connaître les composants React et la manière dont Gatsby les utilise.

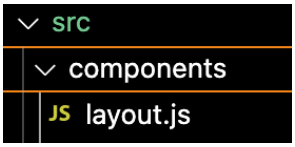
▼ Qu'est-ce que React ?

React est la bibliothèque JavaScript que Gatsby utilise pour créer des interfaces utilisateur (UI). Avec React, on peut décomposer une interface utilisateur en éléments plus petits et réutilisables appelés **components**.

Pour commencer, j'ai créé un composant de page réutilisable : le fichier **layout.js** dans le dossier **components**, qui regroupe tous les éléments partagés à réutiliser sur plusieurs pages. De cette façon, lorsque l'on doit apporter des mises à jour à la

mise en page, on peut effectuer la modification à un seul endroit et elle sera automatiquement appliquée à toutes les pages utilisant ce composant.

Ici, le composant **Layout** contiendra la bar de navigation à afficher sur toutes les pages.



Puis, j'ai créé les pages de mon site, dans le dossier **pages**.

Pour pouvoir être utilisé sur les différentes pages du site, le composent Layout devra être importé au début de chaque page :

```
import Layout from "../components/Layout"
```

sources : https://www.gatsbyjs.com/docs/tutorial/part-2/

Étape 4 : Liaison de l'API GraphQL rickandmortyapi

Pour permettre à un site web Gatsby de communiquer avec l'API GraphQL, il a fallu utiliser un plugin appellé "gatsby-source-graphql".

Ce plugin permet de connecter des API GraphQL arbitraires au GraphQL de Gatsby.

Après l'avoir installé grâce à npm install, dans le terminal : `npm install gatsby-source-graphql`, il est nécessaire de le configurer dans le fichier **gatsby-config.js** :

```
module.exports = {
    siteMetadata: {
        title: `gatsby_api`,
    siteUrl: `https://www.yourdomain.tld`,
```

```
  },
    plugins: [
        `gatsby-plugin-image`,
        `gatsby-plugin-sharp`,
        `gatsby-plugin-sass`,
        `gatsby-transformer-sharp`,
        {
            resolve: "gatsby-source-graphql",
            options: {
                // type de requête de schéma distant
                typeName: "RAM",
                // dossier sous lequel l'API est accessible
                fieldName: "ram",
                // URL à partir de laquelle il faut interroger l'API
                url: "https://rickandmortyapi.com/graphql",
            },
        },
    ],
}
```

Ici, d’autres plugins sont installés :

“gatsby-plugin-image",”gatsby-plugin-sharp”,”gatsby-transformer-sharp” nécessaires pour l’ajout d’image statique à notre site (*voir https://www.gatsbyjs.com/plugins/gatsby-plugin-image/*)

“gatsby-plugin-sass” qui fournit une prise en charge immédiate des feuilles de style Sass/SCSS (*voir https://www.gatsbyjs.com/plugins/gatsby-plugin-sass/*)

Pour vérifier la liaison de l’API à notre site, il faut se rendre sur GraphiQL, via l’adresse : *http://localhost:8000/___graphql*

GraphiQL est un IDE intégré au navigateur, pour explorer les données et le schéma de notre site.

Ici, dans la partie Explorer de GraphiQL, on peut effectivement voir le dossier **ram** (fieldName renseigné dans le fichier **gatsby-config.js**), qui contient les données de l’API rickandmorty.

sources : https://www.gatsbyjs.com/plugins/gatsby-source-graphql/

Étape 5 : Création des templates

L'API rickandmorty ayant beaucoup de données (826 personnages), il m'était impossible de créer manuellement des pages pour chaque personnage.

C'est pour cela qu'il faut créer des templates de pages, pour les pages qui listeront les données (**pageTemplate.js**), ainsi que les pages qui détailleront individuellement chaque personnage (**characterTemplate.js**).

gatsby-node.js :

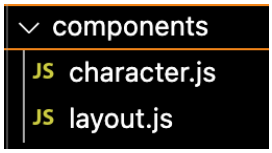
Afin d'interroger nos données pour créer des pages dynamiquement, grâce aux templates, il faut implémenter l'API **createPage** dans le fichier **gatsby-node.js** (à créer à la racine du projet) :

```
const path = require("path")

exports.createPages = async ({ graphql, actions }) => {
    const { createPage } = actions
    const pageTemplate = path.resolve(`src/templates/pageTemplate.js`)
    const characterTemplate = path.resolve(`src/templates/characterTemplate.js`)

    //données nécessaires à la création des pages de la liste
    const data = await graphql(`
    query MyQuery {
      ram {
        characters {
          info {
            pages
          }
          results {
            name
            id
          }
        }
      }
    }
  `)

    const totalPages = data.data.ram.characters.info.pages

    //création des pages de la liste des characters
    for (let page = 1; page <= totalPages; page++) {
        createPage({
            path: `/page/${page}`,
            component: pageTemplate,
            context: {
                page,
                totalPages,
            },
        })
        //données des characters pour remplir les cartes individuelles
        //en fonction de la page
        const characterdata = await graphql(`
        query MyQuery {
          ram {
            characters(page: ${page}) {
              results {
                name
                gender
                id
                image
                location {
                  name
                }
                origin {
                  name
                }
                species
                status
              }
```

```
            }
          }
        }
      `)

        //création des cartes individuelles
        characterdata.data.ram.characters.results.map(character => {
            createPage({
                path: `/character/${character.id}`,
                component: characterTemplate,
                context: {
                    character,
                },
            })
        })
    }
}
```

character.js :

Pour que le fichier **pageTemplate.js** soit plus lisible, j'ai créé un autre component **character.js** dans le dossier **components** qui servira de modèle pour les composants de la liste des personnages.



sources : https://www.gatsbyjs.com/docs/creating-and-modifying-pages/

https://www.gatsbyjs.com/docs/reference/config-files/gatsby-node/#createPages

Étape 6 : Création d'une pagination

Par défaut, GraphQL à une limite de 20 items par page. Pour éviter de n'afficher que les 20 premières entrées, il a fallu mettre en place un système de pagination, pour naviguer entre les différentes pages de l'API.

Pour se faire :

```
const PageTemplate = ({ data }) => {

//total characters
const [characters] = useState(
    data.ram.characters.results
```

```
)
//total pages
const [totalpage] = useState(
  data.ram.characters.info.pages
)
return (
    <Layout>
        <div>
            //affiche carte character
            <div className={containercharacter}>
                <h1 className={tilte}>Character List</h1>
                <Characters characters={characters} />
                <br></br><br></br><br></br>
            </div>
            //création pagination
            <div className={containerpagination}>
                    //création tableau dont la longueur est égale au nb de pages
                    {[...Array(totalpage)].map((_, pageNo) => (
                        <Link
                            className={pagination__pgnumber}
                            to={`/page/${pageNo + 1}`}
                            key={pageNo}
                        >
                            <div>
                                {pageNo + 1}
                            </div>
                        </Link>
                    ))}
                </div>
        </div>
    </Layout>
)
}

export default PageTemplate

//données des characters pour remplir les cartes individuelles
//en fonction de la page
export const query = graphql  `query ($page: Int!) {
ram {
  characters(page: $page) {
  info {
    pages         }
  results {
    id
    name
    gender
    image
    species
    status
    }
  }
}}`
```

source : https://blog.devgenius.io/gatsby-js-pagination-82bef53764cc

Étape 7 : Création d’une barre de recherche

Le principal objectif de ce site était de pouvoir rechercher un personnage de l’API.

J’avais d’abord voulu ajouter une fonction de recherche grâce à Agolia, mais par la suite, j’ai vu qu’on ne pouvait pas faire les requêtes convenablement, car on ne pouvait pas sélectionner uniquement le nom d’un personnage. Dans ce sens, la recherche ne fonctionnait pas elle recherchait aussi le texte entré par l’utilisateur dans l’url des images. (*voir https://www.gatsbyjs.com/docs/adding-search-with-algolia/*)

J’ai donc opté pour une autre solution :

1. Pour commencer, dans le fichier **pageTemplate.js** il faut faire la requête GraphQL qui retourne les personnages selon la page demandée :

```
export const query = graphql`
  query ($page: Int!) {
    ram {
      characters(page: $page) {
        info {
          pages
        }
        results {
          id
          name
          gender
          image
          species
          status
        }
      }
    }
  }
`
```

2. Puis, on pourra accéder à nos personnages comme ceci :

```
const [allcharacters] = useState(
        data.ram.characters.results
    )
```

3. Pour filtrer nos personnages en fonction de ce que l'utilisateur tape, nous aurons besoin d'un texte **input** pour recueillir le texte recherché (à placer dans la partie **return**) :

```
<input
       className={searchBar}
       type="text"
       placeholder="Search"
       onChange={handleInputChange}
//handleInputChange gère les modifictations apportées et filtre les personnages
/>
```

4. On pourra accéder au texte recherché de cette façon :

```
const [searchText] = useState('')
```

5. **Input** est controlé par **useState** (de React) :

```
const [state, setState] = useState({
        filteredData: [],
        query: [searchText],
    })
```

6. 1. Une fois tous ces éléments créés, il ne nous reste qu'à filtrer les personnages.

La fonction **handleInputChange** reçoit un objet **event**. La première chose que nous devons faire est de récupérer sa valeur :

```
//filtre les données
    const handleInputChange = event => {
        //valeur de l'élément ayant déclenché la fonction
```

```
        const query = event.target.value
    }
```

6. 2. Une fois qu’on a la valeur saisie par l'utilisateur, on peut commencer le filtrage.

Pour ce faire, nous allons utiliser la fonction **filter** (**filteredData**) sur notre tableau de personnage. Cette fonction renverra un nouveau tableau contenant uniquement les personnages correspondants à la recherche.

```
//filtre les données
    const handleInputChange = event => {
        //valeur de l'élément ayant déclenché la fonction
        const query = event.target.value
        const characters = data.ram.characters.results || []
        //filtre les données
        const filteredData = characters.filter(data => {
            const { name } = data
            return (
                name.toLowerCase().includes(query.toLowerCase())
            )
        })
    }
```

6. 3. Il ne nous reste qu'une étape pour cette fonction, qui consiste à mettre à jour l'état avec cette nouvelle liste de publications filtrées :

```
//filtre les données
    const handleInputChange = event => {
        //valeur de l'élément ayant déclenché la fonction
        const query = event.target.value
        const characters = data.ram.characters.results || []
        //filtre les données
        const filteredData = characters.filter(data => {
            const { name } = data
            return (
                name.toLowerCase().includes(query.toLowerCase())
            )
        })
        //met à jour l'état avec la nouvelle liste de personnage filtrée
        setState({
            query,
            filteredData,
        })
    }
```

7. Enfin, il ne reste plus qu’à retourner les données filtrées :

```
{characters.map(({id, image, name, gender, species}) => {
  return <span key={id}>
    //component de la liste avec carte individuelle
    <Characters characters={characters} />
</span>
})}
```

sources : https://aboutmonica.com/blog/create-gatsby-blog-search-tutorial/