Rapport

Exercice 1

Après la création de compte puis allé sur la partie de l'api interactive, sur la première requête api, une fois que j'ai rentré un champ puis cliqué sur le bouton try it ou j'obtiens le lien suivant

Request URL

```
https://gateway.marvel.com:443/v1/public/characters?series=2&apikey=
```

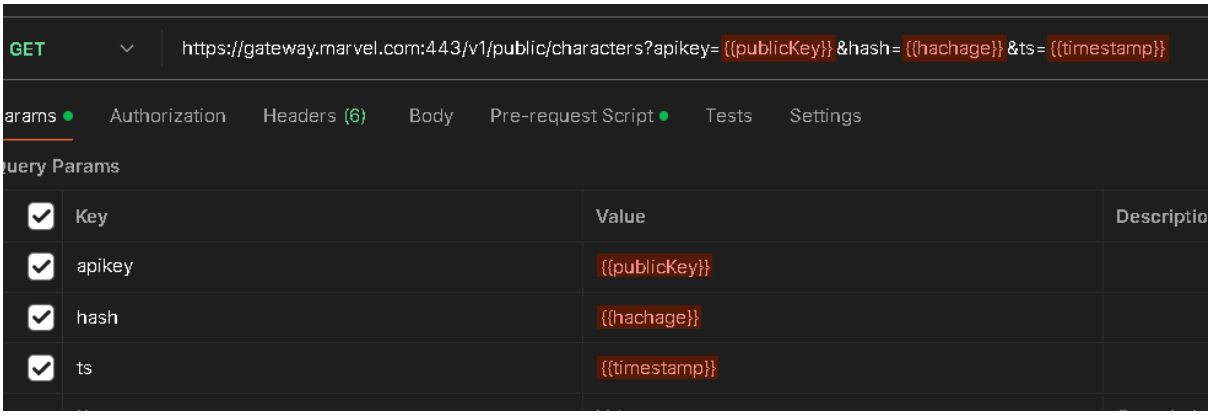
Je peux apercevoir sur ce lien que le champ que j'ai complété est le champ série où j'ai mis la valeur deux qui est représenté par "series=2", que je réalise une requête sur "/v1/public/characters" et que la clé api est vide. Vu que la partie api est vide la réponse reçu est une réponse nous signifiant que notre clé api est invalide

Response Body


```
{
  "code": "InvalidCredentials",
  "message": "The passed API key is invalid."
}
```


Exercice 2

Sur Postman j'ai assigné 3 variables d'environnement, un pour le timestamp, un pour la clé publique et un dernier sur le hash.



Pour le pré-script j'ai initialisé les clés dans les variables publicKey et PrivateKey, j'ai ensuite utilisé la date actuelle pour définir le timestamp que j'ai directement transformé en string derrière. Puis créer le hash selon le string qui est le timestamp + privateKey + publicKey en conformité avec la documentation que j'ai lu. J'ai ensuite remarqué que ceci n'était pas un string et ajouté alors la fonction toString.

J'ai ensuite attribué les valeurs aux variables d'environnements avec la fonction pm.environment.set().

```
3   const ts = new Date().toISOString();
4   const hash = CryptoJS.MD5(ts+privateKey+publicKey);
5
6   pm.environment.set("publicKey", publicKey);
7
8   pm.environment.set("hachage", hash.toString());
9
10  pm.environment.set("timestamp", ts);
```

Après le lancement de la requête je reçois bien la liste de tous les personnages.

Exercice 3

Pour la réalisation de la fonction getHash j'ai simplement réutiliser ce que j'ai pu réaliser dans le TP précédent mais en changeant algorithme de hachage par md5 et utilisé sur le string timestamp + privateKey + publicKey comme à l'exercice précédent.

```
export const getHash = async (publicKey, privateKey, timestamp) : Promise<ArrayBuffer>  => {
    return crypto.createHash('md5').update(timestamp+privateKey+publicKey).digest( algorithm: 'hex');
}
```

Pour la réalisation de la fonction getData j'ai d'abord initialisé les variables nécessaires pour créer le hash, j'ai déplacé les clé en constante du fichier js plutôt qu'initialisé dans la fonction. Le tableau result permettra de stocker le résultat qui sera une liste d'objet json qui aura pour attribut le nom du personnage et son lien vers son portrait. La dernière variable est le timestamp qui correspond simplement à l'heure actuelle.

```
export const getData = async (url) : Promise<json>  => {
    const result : any[]  = [];
    const publicKey : string  = "b466c85dfb561843871a2f024b5e505d";
    const privateKey : string  = "d482ff9ab67ce4bbd83d7f8d0f58879dc2cd753c";
    const ts : string  = new Date().toISOString();
```

J'ajoute ensuite à l'url qu'on a reçu toutes les informations importantes pour la réussite de la requête ainsi qu'une limite de 100 pour récupérer un maximum possible de personnage.

```
url += "?ts=" + ts;
url += "&apikey=" + publicKey;
url += "&hash=" + await getHash(publicKey,privateKey,ts);
url += "&limit=100"
```

J'effectue ensuite la requête à l'api marvel avec la fonction fetch et je transforme le résultat en json. Dans la réponse la liste de personnage se trouve dans l'attribut data qui est aussi un objet json puis dans l'attribut results. Je stocke alors le résultat dans une variable pour faciliter son accès.

```
let response : Response  = await fetch(url);
let data = await response.json();
let characters = data.data.results;
```

Je cherche ensuite alors dans la liste de personnage, avec l'aide d'une boucle forEach, tous les personnages ayant une image, si le lien vers l'image possède image_not_available ceci correspondra à une image avec le texte image not found et non celui du personnage, je vérifie alors simplement si le lien contient cette chaîne de caractères. Si ce n'est pas le cas alors je récupère le nom du personnage ainsi que son image dans je lui rajoute de quoi récupérer son image de type portrait xlarge et je rajoute son extension pour qu'en puisse directement accéder à l'image directement avec le lien. Je stocke ses données dans un objet json que je rajoute dans le tableau de résultat. Le résultat sera ensuite afficher dans la console pour le debug qui sera retiré puis ensuite envoyé.

```
characters.forEach(chara => {
    if(!chara.thumbnail.path.includes("image_not_available")){
        let json : {imageUrl: string, name: any}  = {
            name : chara.name,
            imageUrl : chara.thumbnail.path + "/portrait_xlarge." + chara.thumbnail.extension
        }
        result.push(json)
    }
})

console.log(result)

return result;
```

J'ai remarqué que j'ai oublié de rajouter la description qui était demandé que j'ai rajouté avec la modification suivante :

```
let json : {...}  = {
    name : chara.name,
    desc : chara.description,
    imageUrl : chara.thumbnail.path + "/portrait_xlarge." + chara.thumbnail.extension
}
```

Exercice 4

Pour fastify après avoir ajouté les dépendances nécessaires j'importe les bibliothèque suivante :

```
import fastify from "fastify";
import fastifyView from "@fastify/view";
import handlebars from "handlebars";
import  {getData} from "./api.js";
```

J'initialise ensuite directement le serveur fastify et utilise la fonction register pour pouvoir ajouté handlebars ainsi que le dossier de vue, et le dossier des partials. J'ai trouvé ceci grâce à chatGPT puis compléter le reste avec des ressources sur internet.

```
const app : FastifyInstance<...> & PromiseLike<...>  = fastify();

app.register(fastifyView, {
    engine: {
        handlebars: handlebars,
    },
    templates: '../templates',
    options: {
        partials : {
            header : "../templates/_partials/header.hbs",
            footer : "../templates/_partials/header.hbs",
        }
    }
});
```

Je crée ensuite la page d'accueil du serveur qui affichera l'index. Je passe aussi à cette page la liste des personnages pour qu'il puisse réaliser l'affichage.

```
app.get('/', async (request, reply) => {
    const context = {
        characters: await getData( url: "https://gateway.marvel.com:443/v1/public/characters")
    };
    return reply.view('index.hbs', context);
});
```

Dans index.hbs, j'y attache l'annotation {{> header}} pour rajouter le partial header en début de code et la même chose à la fin pour le footer. J'utilise ensuite l'annotation {{#each characters}} pour réaliser une boucle sur la liste de personnages que j'ai envoyée depuis le serveur. j'y affiche image, nom et description du personnage.

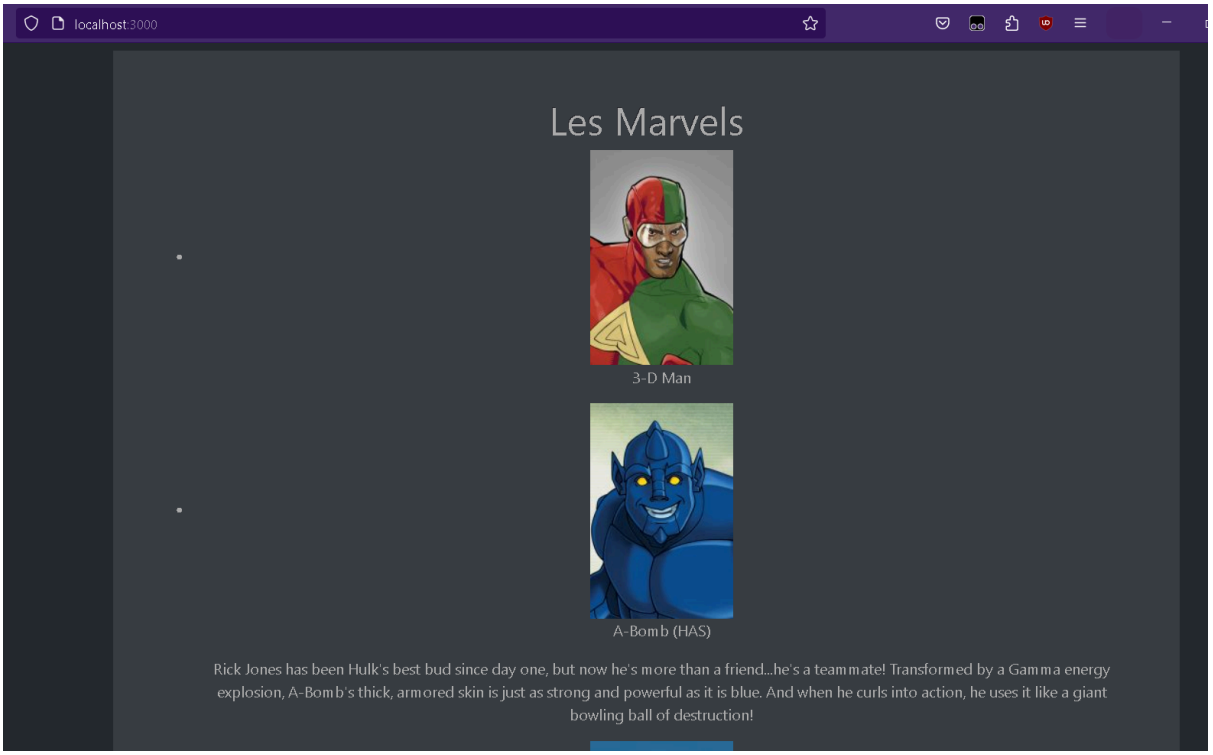
```
{{> header}}
<div class="container-fluid m-2 p-5 bg-primary text-center">
    <h1>Les Marvels</h1>
    <ul>
    {{#each characters}}
      <li>
        <img src ="{{this.imageUrl}}">
        <p>{{this.name}}</p>
        <p>{{this.desc}}</p>
      </li>
    {{/each}}
  </ul>
</div>
<div class="container m-2 p-5 bg-primary  text-center">
    <h2>Mes comics</h2>

</div>
{{> footer}}
```

Enfin je lance le serveur à l'aide de la fonction listen sur le port 3000

```
app.listen( opts: { port: 3000 }, callback: (err) => {
    if (err) throw err;
    console.log(`server listening on ${app.server.address().port}`);
});
```

Après lancement du serveur et accès au site on obtient le résultat suivant :



Exercice 5

Après la création de .dockerignore et du Dockerfile j'ai rajouté dans le fichier docker tout ce qui est nécessaire pour la création de l'image. J'ai d'abord importé l'image lts-bullseye-slim qui est la version minimale d'où le slim ainsi qu'une version avec un long temps de vie signifié par lts (long term service). Je construis ensuite le chemin de fichier avec mkdir, le -p sert à créer les fichiers parents si inexistant puis je change le propriétaire avec chown.

Je spécifie ensuite le répertoire de travail avec WORKDIR et j'y installe les packages dedans avant de lancer l'installation des packages qui ajoute les modules node. Je copie ensuite tout le code source dans ce même répertoire avec .env qui servira plus tard à retirer les variables des clés du code. Puis je lance la commande node src/server.js avec la commande CMD.

```
FROM node:lts-bullseye-slim

RUN mkdir -p home/node/app/node_module
RUN chown node /home/node/app/node_module

WORKDIR home/node/app

COPY package*.json ./

RUN npm install

COPY src ./src

COPY templates ../templates

COPY .env ../

CMD ["node","src/server.js"]
```

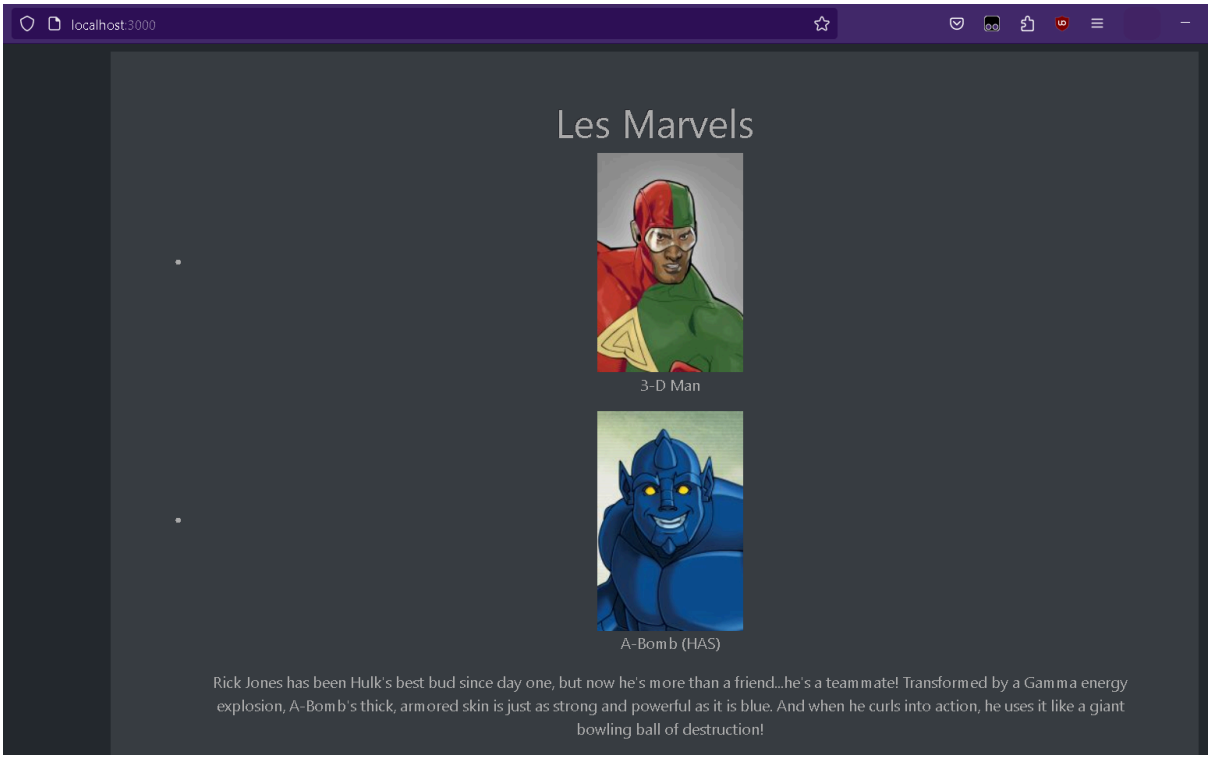
Je peux ensuite créer l'image avec la commande suivante :

```
PS C:\Users\LINos\OneDrive\Bureau\Travaux\3A\Dev avancé\LesMarvels> docker build . -t marvels
```

puis créer le container et le lancer avec la commande suivante :

```
PS C:\Users\LINos\OneDrive\Bureau\Travaux\3A\Dev avancé\LesMarvels> docker run -p 3000:3000 marvels
```

En se dirigeant sur le localhost:3000 j'obtiens bien le même résultat qu'avant



Pour ce qui est de l'extraction des clés j'installe d'abord dotenv avec "npm i dotenv". Je crée ensuite le fichier .env où j'y range mes clés

```
PUBKEY = "b466c85dfb561843871a2f024b5e505d"
PRIKEY = "d482ff9ab67ce4bbd83d7f8d0f58879dc2cd753c"
```

Sur mon api.js, j'import dotenv puis je le configure en lui donnant le chemin relative du fichier api.js à .env.

J'attribue ensuite mes clés avec process.env.leNomDeLaVariableEnvironnement

```
import * as dotenv from "dotenv";

dotenv.config( options: {
    path : "../.env"
})


const publicKey = process.env.PUBKEY;
const privateKey = process.env.PRIKEY;
```

C’est la première que j’ai utilisé docker, j’ai pris alors un certain temps à comprendre l’utilisation mais une fois la prise en main effectué, on remarque facilement que le travail demandé n’est que minime.