

Dossier de Soutenance

Concepteur et développeur d'applications

JANUEL ETIENNE

19 Juillet 2021

Sommaire

Compétences couvertes par le projet	2
Introduction du projet	3
Pour quelle cible ?	3
Quelle concurrence ?	3
Et pour les droits des jeux ?	3
Choix graphiques	3
Pour la sécurité ?	3
Le Projet dans sa globalité	4
Architecture du projet	4
Diagramme UML	5
Spécifications techniques	6
Exemple de Spécification fonctionnelle	7
L'authentification	7
Upload d'images	10
Présentation des jeux d'essais d'une fonctionnalité	14
Schémas de données	15
Envoi de la requête d'ami	15
Acceptation de la requête d'ami	15
Plans de test de l'application	16
Simulation des données	16
Test du controller image	16
Lancement et résultat des tests	18
Préparation au déploiement	19
Préparer les fichiers	19
Mise en place du VPS	19
Transfert des fichiers	19
Veille sur les vulnérabilités de sécurité	20
Les failles exploitables	20
Description d'une situation de travail	21
Veille de l'introduction à NestJS	21
Extrait du site anglophone	21
Traduction	21

Compétences couvertes par le projet

- Concevoir et développer des composants d'interface utilisateur en intégrant les recommandations de sécurité
 - Maquetter une application
 - Développer une interface utilisateur de type desktop
 - Développer des composants d'accès aux données
 - Développer la partie front-end d'une interface utilisateur web
 - Développer la partie back-end d'une interface utilisateur web
- Concevoir et développer la persistance des données en intégrant les recommandations de sécurité
 - Concevoir une base de données
 - Mettre en place une base de données
 - Développer des composants dans le langage d'une base de données
- Concevoir et développer une application multicouche répartie en intégrant les recommandations de sécurité
 - Collaborer à la gestion d'un projet informatique et à l'organisation de l'environnement de développement
 - Concevoir une application
 - Développer des composants métier
 - Construire une application organisée en couches
 - Développer une application mobile
 - Préparer et exécuter les plans de tests d'une application
 - Préparer et exécuter le déploiement d'une application

Résumé du projet

GoBoardGame est une application mobile et web. Elle proposera une variété de jeux de sociétés jouables seuls ou à plusieurs, payants ou gratuits à ses utilisateurs et disposera donc d'une boutique où les utilisateurs pourront y acheter / louer des jeux pour pouvoir y jouer directement depuis n'importe quel ordinateur ou téléphone disposant d'une connexion internet.

Les utilisateurs pourront y gérer leurs profil, personnaliser l'apparence de celui-ci, ajouter des amis et discuter avec eux.

Introduction du projet

Le but de ce projet est d'offrir aux utilisateurs un site web / une application mobile qui contient les jeux de sociétés les plus populaires en version dématérialisée, un environnement social pour jouer entre amis ou seul avec des inconnus ainsi qu'une gamification via un système de succès en jeu.

Pour quelle cible ?

GoBoardGame visera les joueurs dit "casuals", qui lanceront une partie de temps en temps entre amis mais aussi les joueurs plus expérimentés et compétitifs. Les jeux seront à des prix très abordables.

Quelle concurrence ?

Le concurrent principal identifié est BoardGameArena, ils proposent globalement la même chose, c'est-à-dire des jeux de sociétés adaptés au format web jouables à plusieurs mais n'offrent pas grand chose de plus. A l'avenir, goBoardGame penche plus à devenir une plateforme communautaire réunissant des joueurs de jeux de tous les horizons et de les fidéliser en leur offrant un environnement de jeu complet, accessible et agréable.

Et pour les droits des jeux ?

A sa sortie, goBoardGame sera entièrement gratuit et les seuls jeux disponibles seront des jeux libres de droits, par la suite si tout c'est possible, il sera envisagé d'effectuer des démarches auprès des ayants droits afin de négocier ceux-ci.

Choix graphiques

J'ai choisi de réaliser une interface épurée en Material Design, avec des contenus très partitionnés, afin d'améliorer l'expérience utilisateur et qu'il trouve plus facilement ce qu'il souhaite.

Pour la sécurité ?

Concernant la sécurité, je n'ai pas besoin de me protéger du SQL Inject car mes requêtes sont échappées en amont par TypeORM. Aucune donnée de formulaire n'est injectée dans les pages directement, elles passent forcément par l'API qui retournera du coup une erreur si le format des données n'est pas valable.

Le Projet dans sa globalité

Architecture du projet

Le projet sera disponible Navigateur pour le format desktop et via une application pour le mobile. Tous les deux communiquent avec l'API pour traiter les données et avec le serveur WebSocket. L'API est la seule à avoir accès à la base de données. Dans le futur, il est probable que le serveur WebSocket communique également avec l'API afin d'économiser du trafic.

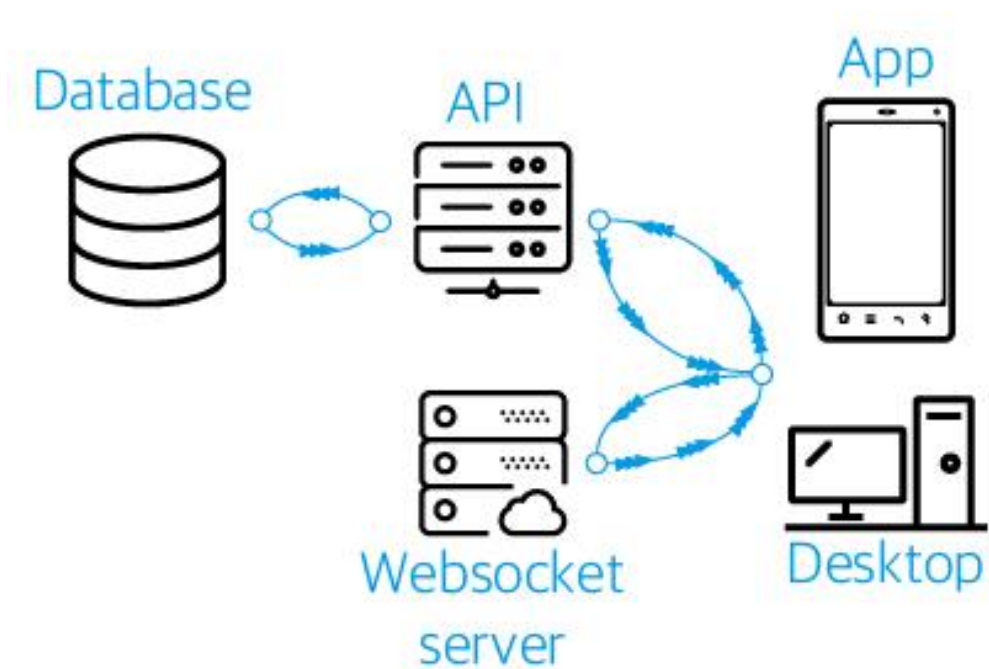
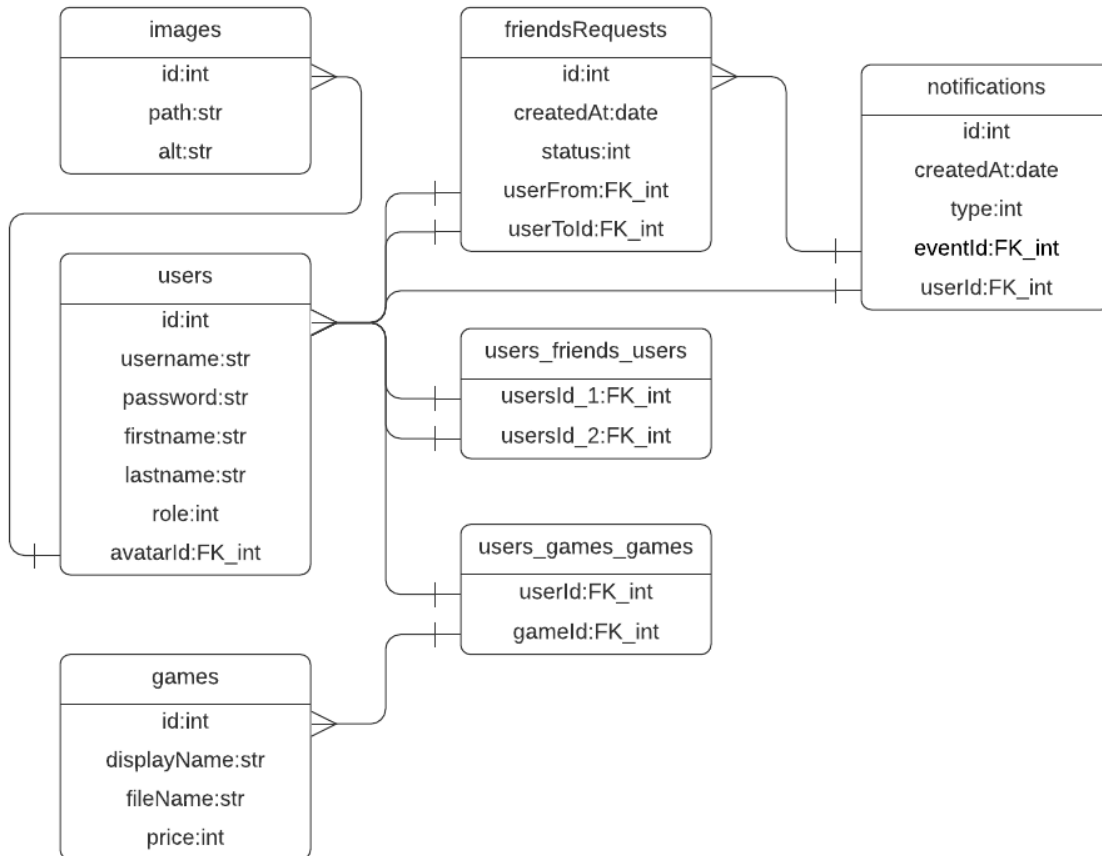


Diagramme UML



Spécifications techniques

- API
 - [NestJS](#)
 - [RxJS](#)
 - [JWT](#) Authentication
 - [TypeOrm](#)
 - [Swagger](#)
- WebSocket Server
 - [nodejs](#)
 - [socket.io](#)
- Database
 - [mysql](#)
- Desktop / Mobile App
 - [ReactJS](#) / [ReactNative](#)
 - [React Router](#)
 - [socket.io](#)
 - [Uifx](#)

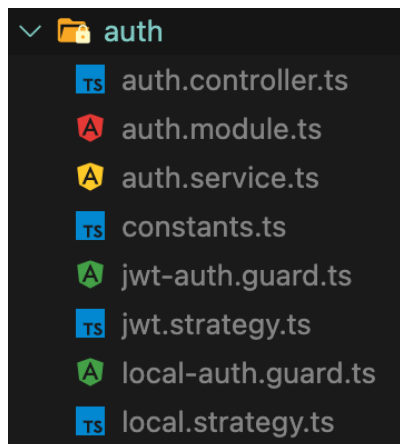
J'ai choisi de privilégier les frameworks javascript pour ce projet afin d'obtenir une certaine cohérence et facilité d'appréhension du projet. L'API quant à elle est en majorité composée de fichiers typescript ce qui aide à la compréhension du projet notamment via l'aide de swagger qui permet d'obtenir une documentation ainsi qu'un outil pour tester les routes et obtenir les schémas de réponses / données.

J'utilise les JWT pour authentifier les utilisateurs, le localStorage du navigateur pour le stocker lui ainsi que diverses données utiles.

Mon serveur WebSocket me permet de gérer diverses fonctionnalités pour les utilisateurs, tels que les notifications, les messages ou les liens d'amitiés. Il servira également à l'avenir pour gérer les parties de jeu.

Exemple de Spécification fonctionnelle

L'authentification



Le processus d'authentification passe par plusieurs logiques / fichiers, grâce notamment à une fonctionnalité de NestJS appelée Guards. Celle-ci permet lorsqu'elle est invoquée via un décorateur d'intercepter la requête et la faire passer par un processus avant qu'elle ne déclenche le processus habituel, si la requête a été validée par notre processus, elle est finalement retournée à la fonction initiale et reprend son cours. Comme expliqué en détails ci dessous :

auth.controller.ts

```
@UseGuards(LocalAuthGuard)
@Post('auth/login')
async login(@Request() req) {
  console.log(req.user);
  return this.authService.login(req.user);
}
```

J'appelle ici le décorateur `@UseGuards()` qui permet d'intercepter la requête avant qu'elle ne soit traitée par la fonction `login()` en lui passant en paramètre un Guard, en l'occurrence je lui passe mon `LocalAuthGuard`.

local-auth.guard.ts

```
@Injectable()
export class LocalAuthGuard extends AuthGuard('local') {}
```

Un guard en NestJS se définit simplement en étendant la classe `AuthGuard` et en lui passant en paramètre le nom du fichier qui contient la stratégie / processus de validation du Guard.

local.strategy.ts

```
@Injectable()
export class LocalStrategy extends PassportStrategy(Strategy) {
  constructor(private authService: AuthService) {
    super();
  }

  async validate(username: string, password: string): Promise<any> {
    const user = await this.authService.validateUser(username, password);
    if (!user) {
      throw new UnauthorizedException();
    }
    return user;
  }
}
```

La stratégie étend la classe PassportStrategy en lui passant en paramètre une stratégie locale qui est instanciée via l'appel de la méthode super() dans le constructeur, ainsi afin de valider le processus, le guard utilisera la fonction validate. Celle-ci récupère les paramètres username et password de la requête par défaut, il est possible de changer les paramètres à récupérer par défaut en passant en paramètre de la méthode super() un objet d'options, dans mon cas, les champs de mon entité user correspondent donc je n'ai pas à le faire.

Ici je stocke le résultat de la fonction asynchrone validateUser dans la variable user.

auth.service.ts

```
async validateUser(username: string, pass: string): Promise<any> {
  const user = await this.usersService.getUserCredentials(username);
  if (user && user.password === pass) {
    const { password, ...result } = user;
    return result;
  }
  return null;
}
```

La fonction validateUser ne fait que récupérer les identifiants, les compare et vérifie que les mots de passe correspondent, si c'est le cas, je crée un objet result et lui assigne les données de user en prenant soin de retirer le mot de passe avant de le retourner.

Finalement la fonction validate vérifie si user est nul, alors elle recrache une 401, sinon elle retourne l'objet user au Guard, qui se charge de peupler la requête interceptée avec les données de l'utilisateur vérifié.

Une fois le Guard validé, la route peut donc reprendre son cours normal et donc exécuter la fonction login du controller.

Mon objet user étant donc validé et présent dans la requête, il ne me reste qu'à générer un jwt content les informations du user et le retourner.

auth.service.ts

```
async login(user: any) {  
  const payload = {  
    sub: user.id,  
    username: user.username,  
    email: user.email,  
    firstname: user.firstname,  
    lastname: user.lastname,  
    role: user.role,  
  };  
  return {  
    access_token: this.jwtService.sign(payload),  
  };  
}
```

La fonction login du AuthService construit un jwt avec les informations de l'utilisateur et retourne un objet contenant un token d'accès ayant pour valeur le jwt encodé.

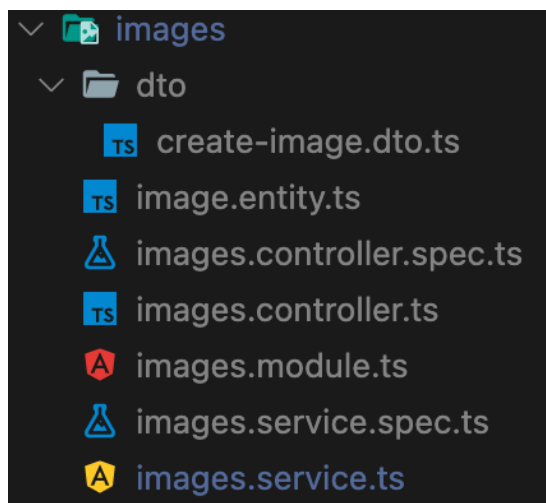
Afin de configurer le temps d'expiration du token et le secret d'encodage, il faut importer le jwtModule dans le authModule qui fait le lien entre tous nos fichiers.

authmodule.ts

```
@Module({  
  imports: [  
    UsersModule,  
    PassportModule,  
    JwtModule.register({  
      secret: jwtConstants.secret,  
      signOptions: { expiresIn: '1d' },  
    }),  
  ],  
  providers: [AuthService, LocalStrategy, JwtStrategy],  
  controllers: [AuthController],  
  exports: [AuthService],  
})  
export class AuthModule {}
```

Upload d'images

Pour la gestion des images, j'ai choisi de m'essayer à quelque chose de différent, puisque mon application comporte beaucoup d'images, je ne souhaite pas saturer ma bande passante en stockant les images sur mon serveur. Afin de soulager celle-ci, j'ai choisi d'utiliser un bucket de google cloud storage pour stocker mes images à distance, et n'avoir qu'à supporter l'envoi des métadonnées de mes images à chaque requête.



Cette fonctionnalité consiste donc à uploader une image sur un bucket google storage public, insérée en base de donnée les métadonnées de notre image ainsi que son url vers le bucket et retourner cette image. Comme expliqué en détails ci dessous :

Afin de récupérer le fichier dans la requête de l'utilisateur, j'utilise le décorateur `@UseInterceptors()` en lui précisant dans ses paramètres les options de mon fichier et son emplacement.

Je récupère les métadonnées de l'image en évoquant le décorateur `@Body` et le stocke dans le paramètre `image` et le décorateur `@UploadedFile` pour le fichier.

J'utilise dans cette fonction des observables et des opérateurs de RxJS, les observables consistent en un flux de données auquel on peut souscrire pour récupérer ces données les unes après les autres, et en les traitant à la volée, ce qui permet d'enchaîner une multitude d'opérations plus facilement et avec plus de maîtrise. Les opérateurs servent à traiter ces observables, `concatMap`, `map` et `catchError` en font partie.

Dans la fonction `createOneImage`, je formate le chemin du fichier afin qu'il soit valide pour l'enregistrement.

image.service.ts

```
@Post()
@UseInterceptors(
  FileInterceptor('file', {
    storage: memoryStorage(),
    limits: {
      fileSize: 5 * 1024 * 1024,
    },
  }),
)
createOneImage(
  @Body() image: Image,
  @UploadedFile() file: Partial<Express.Multer.File>,
): Observable<Image | string> {
  image.path =
    (image.path !== undefined ? image.path + '/' : '') +
    file.originalname.replace(/ /g, '_');
  return this.service.getImageByPath(image.path).pipe(
    map((imageFound) => {
      if (imageFound) {
        throw new ConflictException(
          `The image at path ${image.path} already exists`,
        );
      }
    }),
    concatMap(() => this.service.uploadOneImage(file, image.path)),
    concatMap(() => this.service.create(JSON.parse(JSON.stringify(image)))),
    concatMap(() => {
      return of(`https://storage.googleapis.com/goboardgame/${image.path}`);
    }),
    catchError((err) => {
      if ('sqlMessage' in err) {
        throw new HttpException(err.sqlMessage, 400);
      }
      throw new HttpException(err.response.message, err.response.statusCode);
    }),
  );
}
```

J'appelle ensuite mon premier observable auquel je souscrit via la méthode pipe, celui-ci me retourne une image en fonction de son chemin, puisque mes images sur le bucket google sont identifiées par rapport à leurs chemin je ne risque pas d'avoir de doublon ainsi.

image.service.ts

```
getImageByPath(path: string): Observable<Image | unknown> {  
  return from(this.repository.findOne({ path }));  
}
```

Je récupère le résultat de cet observable avec la méthode map et je recrache une 409 si un objet à été retourné. Sinon je déclenche le second observable.

image.service.ts

```
uploadOneImage = ([  
  file: Partial<Express.Multer.File>,  
  imagePath: string,  
]): Observable<any> =>  
  from(  
    new Promise((resolve, reject) => {  
      const blob = this.bucket.file(imagePath);  
      const blobStream = blob.createWriteStream({  
        resumable: false,  
      });  
  
      blobStream  
        .on('finish', () => {  
          resolve(blob.name);  
        })  
        .on('error', (err) => {  
          reject(err);  
        })  
        .end(file.buffer);  
    }  
  ),  
);
```

Celui-ci se charge d'effectuer l'upload du fichier sur le bucket google via un promise qui est transformée en observable avec l'opérateur from.

Si l'upload s'est effectué, on déclenche alors le dernier observable.

images.controller.ts

```
create(image: Image): Observable<InsertResult> {  
  return from(this.repository.insert(image)).pipe(  
    concatMap((imageInsert) => {  
      console.log(imageInsert);  
      if (!imageInsert) {  
        throw new HttpException(  
          'Something went wrong while persisting image meta-datas',  
          400,  
        );  
      }  
      return of(imageInsert);  
    }  
  ),  
);  
}
```


On insère en base les métadonnées de l'image uploadée sur le bucket et on retourne le résultat de cette opération si l'image à été insérée, sinon on recrache une 400.

Toutes les exceptions qui seront recrachées dans les observables seront interceptées par l'opérateur catchError qui se déclenchera et coupera le flux de l'observable parent.

Le dernier observable est déclenché, celui-ci est une simple string transformée via l'opérateur of contenant l'url publique de l'image uploadée.

Présentation des jeux d'essais d'une fonctionnalité

L'utilisateur John (username: johndoe) souhaite ajouter son ami Jeff (username: jeffbez) dans sa liste d'amis sur goBoardGame.

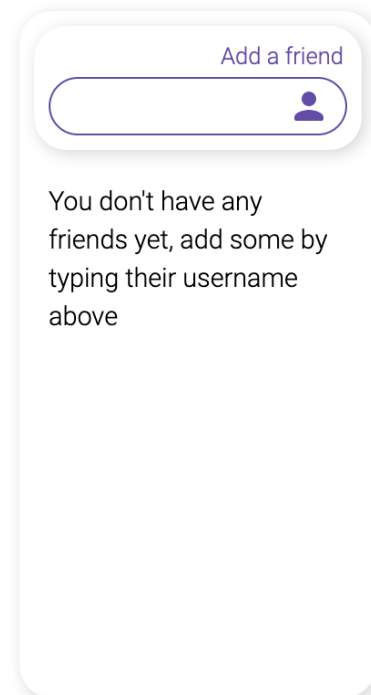
Il clique sur le bouton  pour ouvrir le menu Social, il peut

ensuite taper le nom d'utilisateur de son ami dans le champ en haut du menu, toutes les 0.5s, lorsque l'utilisateur n'écrit plus dans le champ, un appel à l'API est effectué afin de vérifier le nom d'utilisateur, si il n'existe pas déjà de requête d'amis entre ces deux utilisateurs, qu'ils ne sont pas déjà amis et que le nom d'utilisateur existe en base, alors le bouton deviendra vert et l'utilisateur pourra cliquer dessus pour valider sa saisie et envoyer une requête d'ami.

Sinon, il deviendra rouge et affichera l'erreur retournée par l'API en dessous et le submit est désactivé.

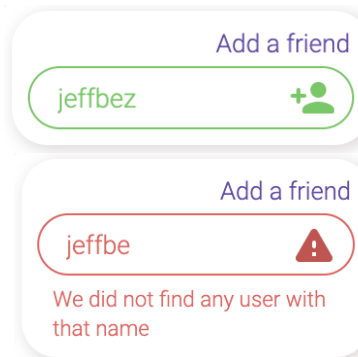
Imaginons que l'utilisateur Jeff soit lui aussi connecté à l'application et attends la requête d'ami de John. Une fois la requête envoyée, Jeff reçoit une notification qui grâce au serveur websocket va actualiser le module de notification en effectuant une requête GET des notifications de cet utilisateur à l'API et déclencher la lecture d'un fichier audio qui fait office de son de notification.

Il apercevra donc le bouton des notifications changer de style, en cliquant dessus, les notifications s'affichent sous la forme de liste, il peut ici accepter ou ignorer l'invitation ou bloquer l'utilisateur qui lui a expédiée.



Add a friend

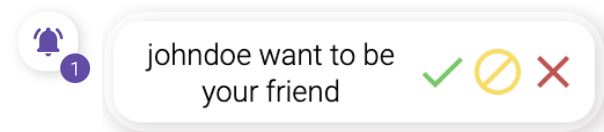
You don't have any friends yet, add some by typing their username above



Add a friend

jeffbez

We did not find any user with that name



1

johndoe want to be your friend

✓ ✗

Une fois que Jeff a accepté la requête, elle se supprime ainsi que la notification liée à cette requête et un lien d'amitié est créé entre les deux utilisateurs. Grâce encore une fois au websocket, on déclenche un événement sur ces deux utilisateurs qui permet de rafraîchir la liste d'amis et celle-ci va donc afficher les données à jour de la liste d'amis des deux utilisateurs.

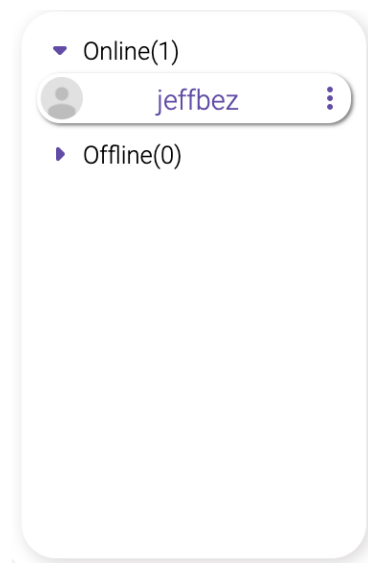
Schémas de données

Envoi de la requête d'ami

- Données envoyées
 - Req Body
 - Partial<user>: { id: int }
 - Req Header
 - Authorization Bearer: accessToken: string
- Données attendues
 - HttpResponse(200 / 409 / 400)

Acceptation de la requête d'ami

- Données envoyées
 - Query Param
 - id: string
 - Req Header
 - Authorization Bearer: accessToken: string
- Données attendues
 - HttpResponse(200 / 409 / 400)



Plans de test de l'application

J'ai uniquement réalisé dans cette application des tests pour la fonctionnalité d'upload d'image par manque de temps, le but étant à terme de viser un code coverage se rapprochant au maximum du 100%, mais surtout de développer les tests en parallèle du développement des fonctionnalités.

J'ai effectué ces tests en utilisant jest, ainsi ils sont exécutables directement depuis l'intérieur du projet et pourront à terme être intégrés dans une potentielle intégration continue.

Simulation des données

Avant de commencer les tests, je vais avoir besoin de simuler les données envoyées par le front lorsqu'il déclenche la route, et cela dans la plupart de mes fichiers de test, au lieu de les réécrire à chaque fois, j'ai créé une classe DataFaker contenant des méthodes afin de générer ces données que tous mes fichiers de tests pourront implémenter.

```
generateFakeFile = (): Partial<Express.Multer.File> => {  
  const file: Partial<Express.Multer.File> = {  
    filename: 'file',  
    originalname: 'fileName.jpg',  
    encoding: '7bit',  
    mimetype: 'image/jpeg',  
    size: 1,  
    stream: new Readable(),  
    buffer: new Buffer(''),  
  };  
  return file;  
};
```

Par exemple dans cette méthode je retourne ce qu'on appelle plus communément un mock partiel de la classe Express.Multer.File

Test du controller image

```
describe('ImageController', () => {  
  let module: TestingModule;  
  let imageController: ImageController;  
  let imageServiceMock: ImageService;  
  let image: Image;  
  let file: Partial<Express.Multer.File>;  
  let dataFaker: DataFaker;  
  
  beforeAll(async () => { ...  
  });  
  
  describe('createOneImage', () => { ...  
  });  
  
  describe('deleteOneImage', () => { ...  
  });  
});
```

Via la fonction describe, je crée un nouveau test global, je lui passe en premier paramètre une chaîne de caractère nommant mon test, et une callback contenant mes tests. Je défini des propriétés auxquelles j'assigne des valeurs dans la fonction beforeAll, celle-ci doit respecter cette nomenclature afin d'être reconnue par jest, elle sera exécutée avant chacun des tests présents dans les fonctions describe suivantes.

```

beforeAll(async () => {
  dataFaker = new DataFaker();
  image = dataFaker.generateFakeImage();
  file = dataFaker.generateFakeFile();

  module = await Test.createTestingModule({
    controllers: [ImageController],
    providers: [
      {
        provide: ImageService,
        useValue: {
          getImageById: jest.fn(),
          getImageByPath: jest.fn(),
          getImages: jest.fn(),
          create: jest.fn(),
          remove: jest.fn(),
        },
      },
    ],
  }).compile();

  imageController = module.get<ImageController>(ImageController);
  imageServiceMock = module.get<ImageService>(ImageService);
});

```

Dans cette fonction, j'instancie donc ma classe DataFaker, que j'utilise directement pour générer les métadonnées et le fichier de l'image.

Dans la variable module, je simule mon module et simule les méthodes de mon service avec la fonction jest.fn().

Je récupère mon controller et mon service depuis le module de test et les attribue aux variables globales de ma classe.

Maintenant que tout mon environnement de test est en place, je vais pouvoir utiliser toutes ces variables et fonctions simulées dans mes tests.

Par exemple, celui-ci à pour but de vérifier qu'un conflit d'exception (409) est bien recraché car l'image uploadée existe déjà.

```

describe('createOneImage', () => {
  it('should throw a ConflictException because image already exists', (done) => {
    jest
      .spyOn(imageServiceMock, 'getImageByPath')
      .mockImplementationOnce(() => of(image));
    imageController
      .createOneImage(image, file)
      .subscribe(fail, (exception) => {
        expect(exception).toBeDefined();
        expect(exception).toBeInstanceOf(HttpException);
        expect(exception.status).toEqual(409);
        expect(exception.message).toEqual(
          `The image at path "${image.path}" already exists`,
        );
        expect(imageServiceMock.getImageByPath).toBeCalledWith(image.path);
        done();
      });
  });
});

```

J'appelle la fonction it qui est un alias de la fonction test, elle prend en paramètre une chaîne de caractère qui définit le nom du test, et un callback qui est appelé à la fin du test.

Via la méthode `spyOn`, je change la valeur de retour de la fonction simulée en un observable de l'objet `image` qui provient lui-même de mon `DataFaker`. Je souscrit ensuite à mon observable et passe en paramètre de mon `subscribe` l'argument `fail`, qui indique à l'observable que je souhaite récupérer l'exception, et en deuxième paramètre, ma `callback` avec en paramètre cette exception. J'exécute via la fonction `expect` des batteries de test en vérifiant que l'exception est bien définie, qu'elle est une instance de la classe `HttpException`, que son message d'erreur est au bon format etc..

Lancement et résultat des tests

Jest dispose d'un CLI qui permet d'exécuter mes tests depuis un terminal via la commande `jest`, j'ai rajoutée celle ci dans mon `package.json` afin de pouvoir la lancer via un alias. Une fois que tous mes tests ont été exécutés, j'obtiens un résumé du déroulement qui affiche les résultats.

```
Test Suites: 3 passed, 3 total
Tests:      18 passed, 18 total
Snapshots:  0 total
Time:       9.212 s, estimated 13 s
Ran all test suites.
🌟 Done in 10.37s.
```

Préparation au déploiement

Dans le cadre du projet professionnel, il est demandé d'effectuer le déploiement d'une application, cependant, celle-ci n'étant pas encore totalement aboutie, je ne peux que préparer celle-ci sans l'effectuer.

Pour cela, je prévois d'héberger l'API et la partie Desktop sur un VPS Debian 10, ainsi que la base de données en MySQL. L'application sera elle déployée sur les différents stores à terme, mais seulement sur le Google Play Store afin d'éviter les problèmes de droits avec l'App Store d'Apple.

Préparer les fichiers

Dans un premier temps, il faudra protéger les variables à risque des fichiers en les transformant en variables d'environnement puis compiler les versions de productions des différents applicatifs.

Mise en place du VPS

Concernant la configuration du VPS, il faut faire en sorte que l'API soit disponible à l'url `api.goboardgame.com` et le desktop sur `goboardgame.com`.

Installer les dépendances des différents applicatifs

Installer un serveur MySQL sans accès distant puisque l'API se trouvera sur le même réseau avec un utilisateur pour l'API.

Mettre en place un pare-feu pour bloquer les ports exploitables avec le package `ufw`.

Installer Apache et configurer les virtual hosts pour faire correspondre les urls.

Créer les fichiers `.htaccess` et les configurer.

Configurer les certificats SSL en utilisant un `certBot`.

Mettre en place un protocole de communication strict avec l'API et les WebSocket afin d'autoriser uniquement les appels depuis le site ou l'application.

Transfert des fichiers

Utiliser un logiciel autorisant le SFTP (Simple File Transfer Protocol) et transférer les fichiers de l'API et de l'application Desktop dans le dossier d'apache de leurs virtual hosts respectifs.

Veille sur les vulnérabilités de sécurité

Les failles exploitables

- SQLInject

Les ORM (Object-Relationnal Mapping) fournissent pour la plupart une surcouche de sécurité afin d'abstraire les requêtes vers SQL et d'exécuter des scripts de sécurité en amont. C'est le cas de TypeORM qui me permet de pas avoir à protéger moi même cette faille tant que j'utilise les méthodes de celui-ci. Cependant, il existe des cas où les méthodes fournies ne suffisaient pas, j'ai donc dû échapper ces requêtes afin d'empêcher l'utilisateur d'exploiter cette vulnérabilité.

- Failles XSS

C'est aussi le cas de ReactJS, celui-ci gère l'affichage des éléments de la page via une méthode `render()` qui compile du JSX en javascript natif, au moment de cette compilation, toutes les données brutes insérées seront échappées et les caractères invalides supprimés. Il faut cependant faire attention à ne pas insérer de données directement dans le DOM via les formulaires afin d'éviter au maximum les potentielles vulnérabilités.

- Modification du JWT / tentative d'usurpation

Un utilisateur malveillant peut essayer de modifier les données de son JWT afin de s'y octroyer un niveau de permission supérieure ou autres. Pour pallier cela, côté API, j'utilise des Guards afin de protéger les routes critiques. Ils permettent d'exécuter au moment où la route est déclenchée un script de vérification en amont, qui va vérifier l'authenticité du JWT et détecter si il a été modifié par l'utilisateur pour finalement retourner une erreur.

Description d'une situation de travail

Veille de l'introduction à NestJS

<https://docs.nestjs.com/guards>

J'ai choisi d'utiliser pour la première fois NestJs sur ce projet et j'ai donc dû apprendre à l'utiliser notamment afin d'appréhender les guards. Voici donc un extrait de la veille effectuée à ce propos.

Extrait du site anglophone

As mentioned, authorization is a great use case for Guards because specific routes should be available only when the caller (usually a specific authenticated user) has sufficient permissions. The AuthGuard that we'll build now assumes an authenticated user (and that, therefore, a token is attached to the request headers). It will extract and validate the token, and use the extracted information to determine whether the request can proceed or not.

The logic inside the `validateRequest()` function can be as simple or sophisticated as needed. The main point of this example is to show how guards fit into the request/response cycle.


Every guard must implement a `canActivate()` function. This function should return a boolean, indicating whether the current request is allowed or not. It can return the response either synchronously or asynchronously (via a Promise or Observable). Nest uses the return value to control the next action:

- if it returns true, the request will be processed.
- if it returns false, Nest will deny the request.

Traduction

Comme mentionné précédemment, l'autorisation d'utilisateurs est un bon cas d'usage pour les Guards, car certaines routes seront disponibles uniquement lorsque l'émetteur (en général un utilisateur authentifié spécifique) aura les permissions nécessaires. L'AuthGuard que nous allons réaliser maintenant dispose d'un utilisateur authentifié (et, par conséquent, d'un token rattaché aux headers de la requête). Il va extraire et valider le token, et utiliser les informations extraites pour déterminer si la requête peut être précédée ou non.

La logique à l'intérieur de la fonction `validateRequest()` peut être aussi compliquée que vous le souhaitez. Le principal objectif de cet exemple est de montrer comment les guards s'intègrent dans le cycle requête / réponse.



Tout Guard doit implémenter la fonction `canActivate()`. Cette fonction doit retourner un booléen, indiquant si la requête actuelle est autorisée ou non. Il peut renvoyer la réponse de manière synchrone ou asynchrone (via une Promise ou un Observable). Nest utilise les valeurs de retour pour contrôler la prochaine action:

- si `true` est retourné, la requête sera procédée.
- si `false` est retourné, Nest rejettera la requête.