RAPPORT D'ACTIVITE DEVOPS

LOGIC@L-CONSEILS

MAI-JUIN 2021

Denis JONGMANEE
EPSI LILLE 2020/2021
PREMIERE ANNEE B1

Remerciements

Tout d’abord, je remercie Logic@l-Conseils et Malik HAMDANI, fondateur de la société et Cécile HADAAD, responsable des ressources humaines de m’avoir accueilli dans leur société pour mon stage de première année DEVOPS.

Je tiens à remercier Ahmed AIT-BOUHOU, mon tuteur de stage pour m’avoir découvert et appris le métier de développeur Full-stack ainsi que les technologies et les notions sur l’architecture des projets.

Je remercie aussi l’EPSI pour m’avoir donné la possibilité d’effectuer ce stage par les connaissances de base d’un développeur et Morgane GROCKOWIAK pour m’avoir répondu à mes questions au niveau administration.

Je remercie également Patrick JONGMANEE pour m’avoir mis en contact avec Malik.

Enfin, je remercie Maxime GAUDUIN, responsable du pôle de développement pour m’avoir consacré du temps au début du stage bien qu’il n’ait pas pu se faire comme il était prévu.

Table des matières

Introduction

Etant en première année B1 pour la formation de DEVOPS en école informatique EPSI Lille, je suis amené à effectuer un stage de fin d'année d'une durée de sept semaines. Mon stage s'est déroulé à Logic@l-Conseils, société de services numériques.

Au cours du stage j'ai pu travailler sur deux projets. Ce document est le rapport d'activité de cette période et comporte plusieurs parties. Le rapport commence par la représentation de ma société qui m'a accueilli. Puis il traitera des détails des deux projets l'un après l'autre (six semaines pour le premier projet et une semaine pour le second). Enfin il se termine par l'explication des réalisations de ce que j'ai pu faire durant mon stage.

1. Présentation de l'entreprise d'accueil et des projets

J'ai effectué mon stage dans la société Logic@l-Conseils. Il s'agit d'une start-up créée en janvier 2018 et fondée par Malik HAMDANI. C'est une société de services numériques. L'entreprise s'opère sur trois pôles d'activité : Le développement, la data-science et le data-management. Pour ma part, j'ai rejoint celui de développement sous la tutelle de Ahmed AIT-BOUHOU, développeur full-stack JAVASCRIPT.

De base, j'étais sous la tutelle de Maxime GAUDUIN, responsable du pôle d'études et de développement pour un projet développé en Python. Cependant, le projet ne s'était pas encore lancé, j'ai donc rejoint Ahmed pour l'assister sur les projets dont il était en charge.

1.1. Le site de pronostics

Ce projet a pour but, aux employés de Logic@l-Conseils, de faire des pronostics entre eux sur l'Euro 2020 (reporté en 2021 de cause de la crise sanitaire). L'entreprise a donc mis un de ses employés sur cette mission pour créer un site web interne dans le but de les divertir. Le site de pronostics avait donc une contrainte qui ne dépendait pas de l'entreprise : le temps de développement devait se terminer avant le début de l'Euro. Le site de pronostics doit permettre aux utilisateurs de prédire quelles équipes atteindraient les phases mais aussi de prédire les résultats des matchs. Un classement se fait en fonction d'un système de points et les trois meilleurs joueurs recevront un prix.

1.2. Le projet de gestion annuelle

Actuellement l'entreprise travaille sur fichier Excel pour générer les fiches de mission de leurs employés pour les prestations. Le projet consiste à pouvoir créer ses fiches depuis un formulaire sur un site web pour l'enregistrer dans une base de données et générer un PDF si besoin.

2. Présentation des activités du stage

Durant tout le long du stage, ma formation sur les technologies et les différentes notions de programmation pour le développement du site Web a été nécessaire pour le bon déroulement. En effet, le projet utilisait de nombreuses technologies et framework dont je n'avais jamais utilisé voire vu auparavant. Mon tuteur était bien sûr présent pour que je puisse lui poser des questions, si besoin, sur ces technologies ou la programmation du projet en lui-même.

2.1. FORMATION

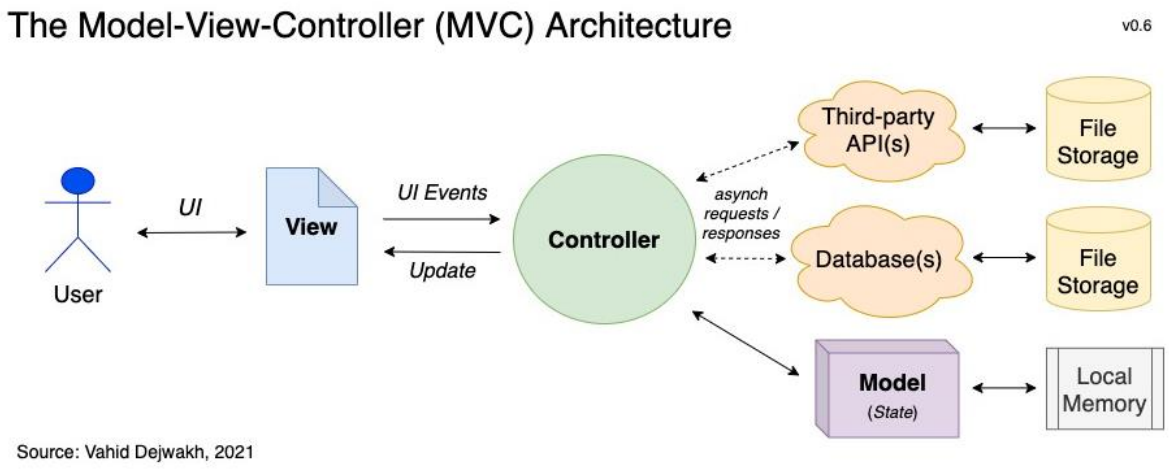
2.1.1. Notion de FULL-STACK : BACKEND/FRONTEND/BASE DE DONNEES

En programmation, on différencie trois couches : le back (site web coté Serveur), le front (site web coté Client) et la base de données. Cette architecture permet d'avoir une structure ordonnée et les différences entre les trois font appel à des compétences de programmation différentes car il s'agit dans la plupart des cas des technologies différentes. D'ailleurs, le back et le front peuvent être coder de manière autonome étant donné que les deux sont considérés comme deux projets distincts. Le Full-stack est le regroupement des deux. Faire du Full-stack revient à codé l'ensemble des couches dans un projet.

Le développement du projet adopte une architecture spécifique et connue par tous les développeurs. Il s'agit de l'architecture MVC.

2.1.2. MVC

L'architecture MVC est utilisée dans ce projet. MVC signifiant Modèle-Vue-Contrôleur permet d'avoir une structure organisée et de séparer les tâches dans les parties d'un programme :



Pour commencer, Le Modèle représente la gestion des données. Il récupère les données stockées dans la base de données auxquels le site aura besoin d'afficher et les envoie au contrôleur.
Ensuite, la Vue a pour rôle d'afficher les pages à l'utilisateur et de récupérer ses demandes. L'utilisateur n'a accès qu'à la vue. De plus, elle se charge seulement d'afficher ce qu'elle reçoit ainsi que de récupérer et envoyer les données entrées par l'utilisateur, elle n'effectue ni traitement ni modification de données.
Et enfin, le Contrôleur fait le lien entre le Modèle et la Vue et se charge du traitement des données si besoin. Il interagit avec la Vue et le Modèle en leur donnant les instructions. Le Modèle ne peut interagir avec la vue qu'indirectement. Il doit forcément passer par le Contrôleur pour échanger ses données.

L’architecture MVC est très adoptée dans de nombreux projets du fait de son efficacité par la séparation des différentes tâches et sa sécurité. Les données sont protégées et sont réellement difficile d’accès par l’utilisateur.

Pour adopter l’architecture MVC, un projet en Javascript peut utiliser le MEAN-stack.

2.1.3. MEAN

Le MEAN est une pile de logiciels codée en javascript pour la création d’application web. Cela permet d’utiliser du javascript dans l’ensemble du programme. C’est une plateforme essentielle pour un développeur Web Full-stack Javascript. Chaque lettre représente une technologie utilisée dans la plateforme : M pour MongoDB, E pour Express, A pour Angular et N pour Node. S’il l’on veut situer Le MEAN dans un développement de projet Full-stack, on aura Angular pour le front, Express et Node pour le back et MongoDB pour la base de données

Les technologies utilisent deux notions importantes dans la programmation : Les interfaces et le Typescript.

2.1.4. INTERFACE

Les interfaces permettent de « créer des types personnalisés ». Ils ne sont pas obligatoires à la programmation mais permettent d’aider le développeur quand il veut typer une variable ou une constance qu’il a besoin.

2.1.5. TYPESCRIPT

L’ensemble du programme est codé en Typescript. Le Typescript permet d’écrire du Javascript mais de manière plus sécurisé et efficace. Le Typescript est un langage typé contrairement au Javascript qui permet au développeur d’éviter des erreurs de script ou des problèmes lors du lancement du code. On a aussi une différence où le Typescript donne accès aux développeurs la notion de classes qui est primordiale dans la programmation orienté objet des sites web. Un script Typescript en soi ne peut être exploité. Il s’agit donc d’écrire un script ou programme en Typescript pour le compiler et générer un code en Javascript afin d’exécuter un projet.

2.1.6. ANGULAR

Angular est un framework de Javascript côté client. La technologie utilise les components, des éléments réutilisables qui possèdent une vue et un traitement de données, les services pour communiquer avec le back et des routages pour se diriger dans le site. Il utilise lui-même une architecture MVC : le Modèle par les services (récupère les données), le Contrôleur par les fichiers Typescript pour récupérer ce que l'on veut et la Vue par les fichiers HTML, soit les Templates pour l'affichage. La force d'Angular est aussi qu'il n'y a pas besoin de recharger la page pour naviguer sur le site web. Cela permet donc d'augmenter la rapidité et l'efficacité de navigation. On obtient alors un site très rapide.

2.1.7. RXJS

RXJS est une librairie de Javacript. Il permet de prendre en compte le délai entre la demande d'une requête et la réponse du serveur. Dans le cas où le front et le back sont codés et considérer de manière indépendante, les données ne sont pas récupérées directement après la demande. RXJS introduit donc le type Observable et ses fonctions. Dans le cas de la non-utilisation de RXJS, les valeurs correspondantes n'auront pas de valeurs. Un objet Observable veut dire que celui-ci attend une valeur jusqu'à le programme récupère une fin de réponse ou rencontre une erreur.

2.1.8. NODEJS – EXPRESS

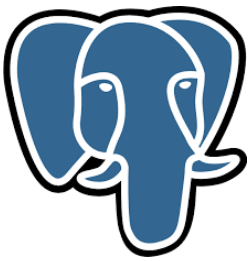
NodeJS est le principal framework de Javascript côté serveur. Il dispose d'un nombre conséquent de modules et de bibliothèques permettant de répondre à de nombreux besoins du développement back-end. Le fait que NodeJS soit aussi intéressant d'utilisation est qu'il dispose de NPM, un gestionnaire de paquets. Et de ce fait, installer un module est facilement accessible (une seule ligne de commande suffit pour installer une librairie).

NodeJs est également utilisé pour son framework Express permettant de créer des applications web côté serveur. Il s'agit d'un framework simple d'utilisation.

2.1.9. ASYNC

Les fonctions sont soit synchrones soit asynchrones. De base les fonctions sont synchrones. Définir une fonction étant asynchrone permet aux programmes d'exécuter plusieurs lignes de code en même temps. Cela permet d'augmenter l'efficacité et la rapidité d'exécution du code. Dans le cas où la ligne de code est nécessaire pour la suite du programme, on peut ajouter un await qui permet d'attendre que la tâche soit finie pour continuer dans la suite.

2.1.10. POSTGRESQL



Durant mon premier projet, j'ai utilisé une base de données relationnelles qui est PostgreSQL. Il s'agit d'une base de données similaire à MySQL (étudié durant l'année d'étude).

2.1.11. NoSQL et MongoDB

Durant cette année d'étude, nous avons appris à utiliser des bases de données relationnelles. Cependant, il existe d'autres types de base. NoSQL est l'un de ces types. Contrairement aux bases de données relationnelles, NoSQL a plus de liberté en termes de contraintes car celui-ci donne plus de liberté en données pour augmenter l'efficacité de stockage. Durant mon stage, j'ai utilisé ce nouveau type de base de données par la base de données MongoDB.

Contrairement aux tables utilisées dans des bases de données relationnelles, MongoDB est orienté Document. C'est-à-dire qu'il stocke les données dans des collections. Par exemple, une collection représentant une bibliothèque contenant des livres avec leur titre et année de parution. Un livre représenterait un document. Il existe des liens entre les documents mais les contraintes ne sont pas fortes.

2.1.12. Versionning

Tout du long du stage, j'ai travaillé avec un outil de gestion de projet. Il s'agit de Gitlab, un outil de versionning. Celui-ci permet de partager les programmes entre développeurs et gère également les conflits, ce qui peut être primordial lorsque deux personnes travaillent séparément sur un même fichier.

Gitlab propose, comme de nombreux outils de gestions de projet, un système de branches qui consiste à définir une arborescence. La première branche se nomme master et représente le programme en production. La deuxième branche se nommant develop est utilisée par les développeurs pour apporter des améliorations et changements dans le programme sans impacter la version opérationnelle du projet (master). Les développeurs ont la possibilité de créer d'autres branches héritées de develop pour pouvoir travailler sur plusieurs fonctionnalités en même temps. Ainsi, grâce à ce système, j'ai pu travailler sur le même fichier que celui de mon tuteur.
Lorsque la fonctionnalité est opérationnelle, la branche est fusionnée avec develop pour appliquer le changement tout tant gérant les conflits.

2.2. LE DEVELOPPEMENT DU SITE WEB DE PRONOSTICS EURO 2020

2.2.1. LA BASE DE DONNEES

PostgreSQL et PgAdmin (interface graphique pour gérer les bases) ont été utilisés pour la partie base de données.

L'une de mes premières tâches, pendant ma formation, consistait à récupérer les logos des équipes de football pour les utilisait dans le site web. Il est très couteux en termes de performance de stocker des images dans la base de données. Il est donc mieux de stocker les images dans un dossier et de récupérer le nom des fichiers (images) dans la base de données. Après avoir fait attention à la pertinence des logos, c'est-à-dire de prendre la dernière version des logos des équipes, il m'a fallu les renommer par un nom de convention : logo_equipe.png. La convention des noms de fichier veut que les caractères soient en minuscule et que les espaces sont représentées par « _ ». Le mieux est d'utiliser des images avec une taille le plus grand possible pour garder la qualité lorsque les images sont réduites. Le projet dispose également d'un dossier de logo de taille plus petite obtenus par un logiciel de redimensionnement trouvé sur internet car il est possible que la conversion soit meilleure par un logiciel annexe plutôt que le navigateur en lui-même. Le premier site de redimensionnement d'image trouvé a été le choix pour ce nouveau dossier car après avoir vu le résultat, celui-ci était acceptable.

Durant mon stage, j'avais la charge de quelques améliorations et d'ajouts de la base de données. Il y a l'amélioration de la création d'un compte. En effet, pour avoir des comptes utilisateur avec des adresses mails vraiment existants, il a fallu coder une validation de compte avant la possibilité de se connecter. Les champs « email_hash » et « email_activated » ont été ajouté à la table « members ». « email_activated » est un champ booléen qui permet de savoir si le compte est activé. Quant à « email_hash », il s'agit juste de l'adresse mail de l'utilisateur mais cryptée. Il servait de clé pour activer le compte en comparant « email_hash » et l'adresse mail cryptée.

Il y a eu ensuite l'ajout de champ pour la table « bet_teams » qui contenait les paris de joueurs sur les équipes. Il a fallu ajouter trois champs à cette table : score, is_done, is_won.

Voici le script SQL :

```
ALTER TABLE public.bet_teams
        ADD COLUMN score smallint default(0);
ALTER TABLE public.bet_teams
        ADD COLUMN is_done boolean default(false);
ALTER TABLE public.bet_teams
        ADD COLUMN is_won boolean default(false);
```

« ALTER TABLE » permet de modifier une table déjà existante. Ce code peut être amélioré pour avoir un seul ALTER TABLE pour les trois champs.

Je devais ensuite créer un script SQL qui permet de créer des procédures stockées. Il s'agit des fonctions intégrées dans la base de données et permet d'améliorer l'efficacité du code en évitant des aller-retours entre la base de données et le back. Cette méthode prend cependant plus de mémoire étant stockée.

```
CREATE FUNCTION public.somme_score(_id integer) RETURNS integer
AS $$
SELECT SUM(score) as _somme
FROM bet_teams
WHERE is_done='true' AND is_won='true' AND user_id=_id;
$$;
LANGUAGE SQL
RETURNS NULL ON NULL INPUT;
```

Cette fonction stockée permet d'avoir le score d'un utilisateur à partir de ses pronostics sur les équipes.

Ensuite il a fallu ajouter, dans la table « rounds » qui représente les phases du tournois, le nombre de pronostics d'équipe maximale par phase. L'utilisateur a la possibilité de seize prédictions au huitième de finale, huit en quart de finale, quatre en demi-finale et deux en finale.

Il y a eu ensuite, similaire à la table « bet_teams », la création de la table « bet_matchs ». Il s'agit des pronostics sur les matchs (le deuxième type de pronostics).

```
CREATE TABLE public.bet-matchs
(
id serial NOT NULL,
user_id int,
match_id integer,
created_date timestamp with time zone DEFAULT now(),
team_id integer,
is_done boolean DEFAULT false,
is_ok boolean DEFAULT false,
score integer,
CONSTRAINT bet_matchs_pkey PRIMARY KEY (user_id,match_id)
)
WITH (
OIDS = FALSE
)
TABLESPACE pg_default;
ALTER TABLE public.bet_matchs
OWNER to postgres;
```

Et enfin, des triggers ont aussi été utilisé dans ce projet. Les triggers sont des script SQL qui sont exécutés seulement lorsqu'un certain type de requête survient. Dans le cas du projet, il s'agit de mettre à jour les données de pronostics et le score

des utilisateurs lorsque l'utilisateur lui-même ajoute un pari ou lorsqu'on obtient le résultat d'un match. Un trigger possède deux parties : la première partie s'agit d'écrire ce qu'il faut faire lorsqu'un trigger est activé. La deuxième permet de savoir quand exécuter le trigger. Un trigger ne s'effectue qu'au moment d'un DELETE, UPDATE ou/et INSERT d'une table spécifiée dans le script. Il est possible de lancer un trigger avant ou après la requête d'activation. Un exemple de trigger utilisé dans le projet :

1ère partie du trigger :

```
CREATE FUNCTION update_member_bet_teams() returns TRIGGER
	LANGUAGE PLPGSQL
	as $$
		BEGIN
		IF TG_OP = 'DELETE' THEN
			UPDATE members
			SET balance =
sum_score_bet_teams(OLD."user_id")+sum_score_bet_matchs(OLD."user_id"),
			total_bets_teams = count_total_bet_teams(OLD."user_id")
			WHERE id=OLD."user_id";

		ELSIF TG_OP = 'UPDATE' THEN
			UPDATE members
			SET balance =
sum_score_bet_teams(NEW."user_id")+sum_score_bet_matchs(NEW."user_id") ,
			total_successful_bets_teams =
count_total_successful_bet_teams(NEW."user_id")
			WHERE id=NEW."user_id";

		ELSIF TG_OP = 'INSERT' THEN
			UPDATE members
			SET balance = sum_score_bet_teams(NEW."user_id") +
sum_score_bet_matchs(NEW."user_id"),
			total_bets_teams = count_total_bet_teams(NEW."user_id")
			WHERE id=NEW."user_id";
		END IF;
		RETURN NULL;
		END;
	$$;
```

2ème partie du trigger :

```
CREATE TRIGGER after_insert_update_delete_bet_teams
	AFTER INSERT OR UPDATE OR DELETE ON bet_teams
	FOR EACH ROW
	EXECUTE PROCEDURE update_member_bet_teams();
```

TG_OP permet de savoir quel type d'exécution est en cours. Le trigger peut aussi utiliser des procédures et/ou fonctions stockées.

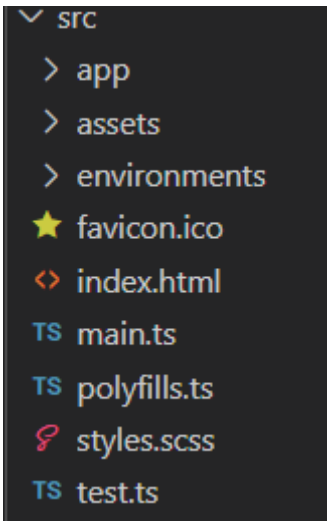
Il m'a fallu créer une requête SQL qui permettait de créer des données et que si les clés primaires sont déjà utilisées de faire une mise à jour au lieu de la création. Voici le script :

```
const q = `INSERT INTO bet_matchs(match_id, user_id, team_id, score) VALUES
(${bet.matchId},${bet.userId},${bet.teamId},${bet.score})
      ON CONFLICT (match_id, user_id, team_id) DO UPDATE SET score =
${bet.score};`
```

Dans le cas où le groupe (match_id, user_id, team_id) est déjà présent dans la table « bet_matchs », on effectue seulement une modification sur « score ».

2.2.2. LE FRONT

Le front est entièrement codé en Angular, un framework de Javascript. Angular est facile à installer car il suffit d'installer le CLI d'Angular et de lancer la commande « ng new » pour créer un nouveau projet. Lorsqu'un nouveau projet est généré, un nouveau répertoire est créé et tout est déjà prêt pour commencer à coder. Il reste juste à établir les configurations dans les fichiers dédiés pour adapter le projet à ceux que l'on veut. Bien que de nombreuses librairies sont déjà installées lors de la génération du projet, tout comme NodeJS, Angular dispose de NPM, distributeur de paquets pour installer les modules ou les librairies manquants si besoin.



Le dossier du projet contient un dossier src qui permet aux développeurs d'écrire le code créant le site web. Le code pour le développement du site se trouve généralement dans App.

Angular utilise la technologie des components qui permet d'avoir une vue réutilisable. Par exemple, chaque page du site utilise la même tête (header) et pied (footer). Ce sont tous les deux des components. Chaque component est composé d'un fichier HTML, un fichier Typescript et un fichier SCSS (fichier de style selon la configuration du projet et qui permet d'appliquer un CSS propre au component). Le fichier Typescript est une classe où sont représentées les données du component et le HTML la partie visuelle.

Le template général du site web est un template acheté qui permet d'avoir un CSS en lien avec le thème du site : le football et l'euro 2020.

Pour installer l'environnement sur mon poste de travail, je devais installer angular et récupérer par gitlab le projet créé par mon tuteur. La commande « ng serve » sert alors à lancer le code et le serveur sur le réseau local au port 4200. (port d'angular par défaut). Le code Typescript est alors compilé et génère un code en Javascript qui est aussitôt lancé automatiquement.

Lors du stage, dans mes premières tâches du front, j'ai pu améliorer un component en créant un décompte dynamique qui montre le temps réel restant avant le début du prochain match. L'entière tâche devait se faire uniquement sur le front.

Après avoir vu une documentation, j'ai proposé à mon tuteur une solution. Cette solution consistait à calculer la différence entre la date du match et la date du jour actuel et de stocker chacun des 4 valeurs dans un attribut différent pour les afficher. Cette opération se faisait tous les secondes grâce à une fonction implémentée dans Angular. Cela fonctionnait mais elle n'était pas optimisée et n'utilisait pas les ressources d'Angular. En effet, Angular dispose de fonctions de filtre. Le filtre est représenté par « | ». « upCommingDate » correspond à l'attribut qui contient le décompte. Les attributs du fichier Typescript peut être afficher grâce aux doubles accolades {{ }}.

```
<div class="single-time">
    <span class="number hour">{{upCommingDate | date: 'HH'}}</span>
    <span class="title">Hours</span>
</div>
<div class="single-time">
    <span class="number minute">{{upCommingDate | date: 'mm'}}</span>
    <span class="title">Minutes</span>
</div>
<div class="single-time">
    <span class="number second">{{upCommingDate | date: 'ss'}}</span>
    <span class="title">Seconds</span>
</div>
```

Cette solution permet donc d'avoir une seule variable pour le décompte au lieu de 4.

Par la suite, ma mission était de créer un lien d'ancrage sur un bouton vers une nouvelle page.

Pour cela, Angular utilise un système de routage qui permet à l'utilisateur de naviguer entre les pages. Un component peut être utilisé en tant qu'une page du site.

Elle a un impact sur les liens d'ancrages (liens hypertextes). En effet, dans le HTML habituel, on utilise « href » où il faut spécifier le chemin de la page. Dans Angular on utilise « [routerLink] » correspondant au chemin défini dans le fichier de routage.

```
const routes: Routes = [
  { path: '', component: LoginComponent },
  { path: 'login', component: LoginComponent },
  { path: 'register', component: RegisterComponent },
```

Par exemple, pour accéder à la page d'inscription, il suffit d'accéder à monsiteweb/register pour afficher le component Register.

Une autre mission était de créer une page de profil où l'utilisateur a la possibilité de changer son nom, prénom, sexe et son mot de passe.
Pour coder la page de profil, je me suis inspiré de la page d'inscription créée par mon tuteur car elle est très similaire avec une seule différence. Il s'agit d'afficher un formulaire avec les champs déjà rempli par les informations du profil. Pour cela, il m'a fallu créer un component.

Pour créer un component, il suffit d'entrer la ligne de commande :
« ng generate component account/my-profile » ou « ng g c account/myprofile » pour faire plus court.

On peut ajouter « --skipt-test » pour ne pas créer un fichier de test ou « --dry-run » pour voir le résultat de la commande pour être sûr de la création du component. Cette ligne de commande va créer les components dans le dossier account se trouvant dans src/app (il crée le dossier si celui-ci n'existe pas) et en créant un dossier « my-profile » qui contient trois fichiers et mettre à jour le fichier « account.module.ts » qui contient les déclaration de tous les modules et components du dossier account :

```
CREATE src/app/account/my-profile/my-profile.component.html (25 bytes)
CREATE src/app/account/my-profile/my-profile.component.ts (291 bytes)
CREATE src/app/account/my-profile/my-profile.component.scss (0 bytes)
UPDATE src/app/account/account.module.ts (670 bytes)
```

Dans la partie HTML, il y a les deux formulaires (modification du profil et modification du mot de passe). Angular introduit la notion de ngModel, un module ancré qui permet de récupérer les valeurs des champs et l'introduire sous la forme d'un objet. Il faut ensuite créer un service qui se chargera de récupérer cet objet et de l'envoyer vers le back. Pour cela, on dispose du module HttpClient qui permet de faire des requêtes vers une API. Il existe quatre types de requêtes :

- Get permet de demander la récupération de données
- Post permet de créer une nouvelle donnée ou de la réécrire complétement
- Put pour modifier des données
- Delete pour supprimer

Il faut bien-sûr spécifier l'adresse du back et un objet qui contient les données si besoin. La création d'un service est similaire à la création d'un component :
« ng g service nom_service » ou en plus simple « ng g s nom_service ».

Les services sont des classes utilisables n'importe où et ont une extension .service.ts Les services sont regroupés dans un même fichier et chaque service représente un type de données de la base (ici une table). De ce fait, la page de profil va faire appel au service account.
Pour utiliser un module dans un service ou un component, il faut le déclarer dans son constructeur en privé.

Spécifié @Component permet d'affirmer que la classe est un component.

```
@Component({
  selector: 'app-my-profile',
  templateUrl: './my-profile.component.html',
  styleUrls: ['./my-profile.component.scss'],
})
export class MyProfileComponent implements OnInit, OnDestroy {
```

Spécifié @Injectable permet d'affirmer que la classe est un service.

```
@Injectable({
  providedIn: 'root'
})
export class MemberService {

  constructor(private http: HttpClient) { }
```

Pour faire appel au module déclaré dans le constructeur, il faut utiliser « this », similaire à PHP et équivalent à « self » dans Python qui permet de spécifier l'objet actuel auquel on travaille.

Un component dispose d'une méthode « ngOnInit() » qui se lance automatiquement lorsque qu'il est généré. Cette méthode permet donc de récupérer les données qui viennent du back pour les afficher sur la page. Il fallait donc récupérer le nom, le prénom et le sexe de la personne par le service et de les afficher dans les bons champs. C'est ngModel qui permet cela car en initialisant l'objet au démarrage de la page, elle affiche et récupére directement les données. Concernant le sexe du profil, le type du champ choisi favorise un menu déroulant au lieu de boutons par choix de mon tuteur.

Quand la partie fonctionnalité de la page était terminée, il fallait instaurer des contrôles sur les formulaires pour éviter des conflits avec la base de données et de s'assurer de l'authenticité des informations voulues par l'utilisateur. C'est-à-dire de vérifier si les champs ne sont pas vides.

Concernant la modification de mot de passe, l'utilisateur doit entrer son mot de passe actuel et de confirmer le nouveau. Il fallait donc vérifier qu'il s'agit bien du mot de passe actuel de l'utilisateur (pour vérifier qu'il s'agit bien de l'utilisateur qui souhaite le changer) et que les deux champs correspondant au nouveau mot de

passe sont les mêmes. Pour cela, on préfère vérifier seulement côté front pour éviter des aller-retours entre le front et le back et éviter de diminuer les performances du site. ngModel dispose de méthodes pour vérifier les champs. Lorsque l'utilisateur clique sur le bouton de modification du profil, on vérifie les champs en question et regarde si tous les contrôles sont positifs. Si c'est le cas les nouvelles informations sont envoyés au back. Il existe des contrôles pour savoir si un champ est bien rempli. Mais s'il n'y a pas de contrôles spécifiques, il est possible d'en créer des personnalisés.

```
if (this._modelNewPassword.cpassword !== this._modelNewPassword.newPassword) {
  form.controls['cpassword'].setErrors({ 'noMatch': true });
  form.controls['cpassword'].markAsTouched();

  this._toastrService.warning('Les deux mots de passe ne sont pas identiques');
  return;
}
```

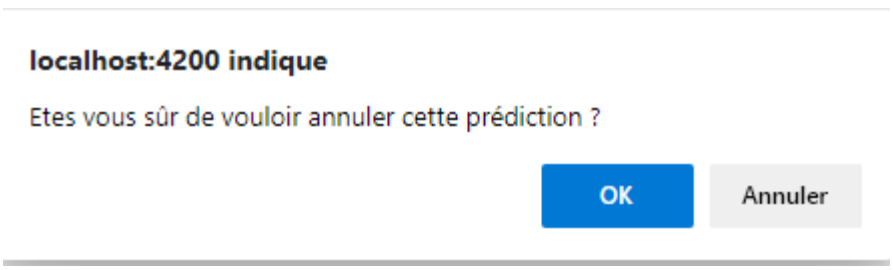
Au lieu d'afficher du texte sur la page pour communiquer avec l'utilisateur (confirmation du changement, problème de formulaire, problème avec les données entrées…), le site utilise des messages d'alerte grâce au module ToastrModule. Il dispose du service ToastrService pour afficher des messages lorsqu'un évènement survient.

On utilise trois messages d'alerte différents :

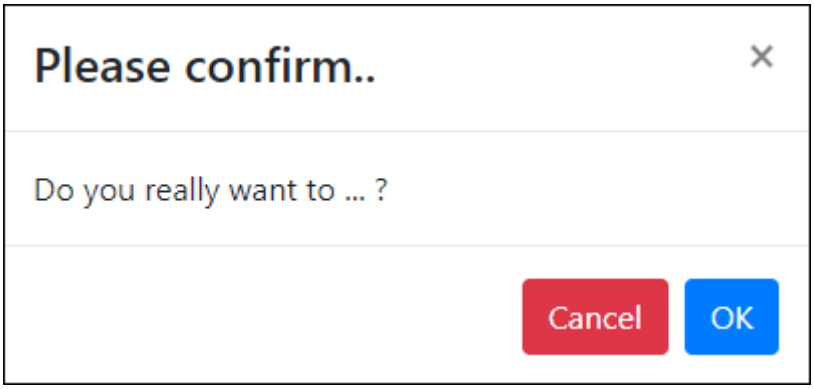
- Alerte vert (succès)
- Alerte orange (problème connu)
- Alerte rouge (problème inconnu)

La page de profil contient aussi une amélioration codée plus tardivement qui est la création d'un champ pour l'image du profil. L'utilisateur a le choix entre plusieurs images pour illustrer son profil. Il peut le choisir dans la page de profil. Les images sont stockées directement dans le dossier source du projet.

Les utilisateurs ont la possibilité de nommer des équipes qui pourraient passer les différentes phases à partir des huitièmes de finale. Ma tâche consistait à créer une validation de suppression d'un pronostic sur le choix d'une équipe. Pour cela, j'ai utilisé une fonction alert(), fonction de javascript où une boîte de dialogue apparaissait pour que l'utilisateur puisse confirmer son choix de suppression.



Mon tuteur voulait cependant une fenêtre de dialogue avec un meilleur aspect visuel et m'a fait part d'un lien d'un projet open source sur StackBlitz (voir annexe pour le lien) qui propose un component d'Angular qui sert de boîte de dialogue.



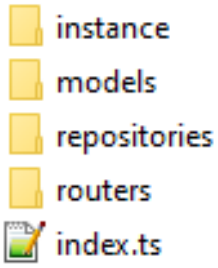
Il s'agit en réalité d'un component. J'ai donc dû le comprendre et l'introduire au projet pour l'adapter dans mon code.

Il m'a fallu mettre à jour une page qui doit contenir des données dynamiques depuis la base de données. Il s'agit de la page « dashboard » qui permet à l'utilisateur de voir ses statistiques : son score, son nombre de pronostics, son nombre de pronostics gagnantes et son classement. L'ensemble des données qui doivent être affichée est directement récupérer du back. Dans une première solution, j'ai récupéré toutes les données sauf le classement (une donnée non présente dans la base de données et doit être retrouvée) en créant un filtre personnalisé et récupérant l'ensemble des points des joueurs inscrits sur la plateforme. L'affichage devait donc apparaitre l'ensemble des joueurs rangés par point et d'afficher son index dans la liste. En appliquant le filtre, on récupère seulement l'utilisateur. Cependant en termes de performance, cette solution n'est pas optimale car le front fait du traitement et la partie client doit éviter de faire d'en faire pour garder la rapidité de navigation. Le back devait donc s'en charger et par le service, récupérer exactement les données souhaitées.

Le programme utilisait une API pour mettre à jour sa base de données concernant les résultats des matchs. Cependant, mon tuteur a trouvé que les APIs sur les résultats de l'Euro n'étaient pas assez rapides car la mise à jour des données n'est pas instantanée après la fin du match et de ce fait, la mise à jour des points des pronostics n'étaient pas assez rapides. Il m'a alors demandé de créer une interface administrateur qui permet de choisir un match pas encore fini et d'y entrer les résultats. Un menu déroulant contenait l'ensemble des matchs qui n'étaient pas validés dans la base de données. En choisissant un match, le logo et le nom de l'équipe apparait sur l'écran ainsi que deux champs pour entrer les scores.

2.2.3. LE BACK

Le back est développé sous NodeJS avec son framework Express. Il est codé en utilisant des routeurs pour se diriger dans le programme. Il est organisé sous quatre dossiers et un fichier index.ts. Lors de la demande d'une requête du front. La requête passe par index.ts où on définit le chemin

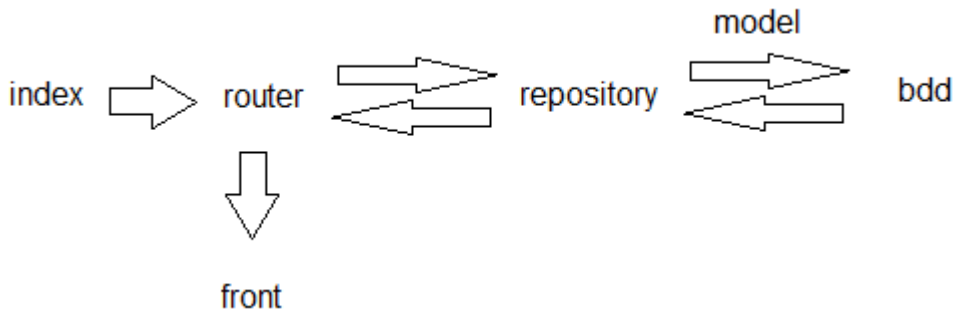



```
app.use("/api/team", teamsRouter);
app.use("/api/teams", teamsRouter);
app.use("/api/group", groupsRouter);
app.use("/api/match", matchsRouter);
app.use("/api/result", resultsRouter);
```

Selon le chemin de la requête, on se dirige vers un routeur spécifique. Il existe autant de router que de tables dans la base de données.

```
export const resultsRouter = express.Router();

resultsRouter.get("/", async (req: Request, res: Response) => {
    try {
        const repository = new MatchRepository();
        const results = await repository.getResults();
        res.status(200).send(results);
    } catch (e) {
        res.status(500).send(e.message);
    }
});
```

Selon le type de la requête HTTP et le chemin associé, on peut définir ce qu'on souhaite faire. Par exemple, ici on se retrouve dans le router des matchs et on reçoit une demande de type « get ». On récupère alors les résultats en questionnant la base de données pour les envoyer au front. On aura alors un parcourt de ce type :



Le modèle dépend de la base de données. Dans ce projet, la base de données étant relationnelle, le module « Sequelize » a été utilisé. Ce module est très

simple d'utilisation, il suffit de définir chaque champ des tables par son type et il est aussi possible de changer son nom dans l'appellation dans le code par un alias.

Afin d'éviter la perte de compte, j'ai eu la charge de créer une page du site qui permet de générer un mot de passe. En effet, les mots de passe étant cryptées, il était impossible de retrouver les mots de passe même aillant accès à la base de données car le cryptage utilisé est du type non bijectif.
Les cryptages bijectifs sont des cryptages où un code non crypté aura forcément un seul résultat de cryptage. De ce fait si on possède la clé, on peut retrouver ce qu'il a été crypté car un décryptage aura forcément un seul résultat. Concernant le cryptage non bijectif où le code crypté est unique. Par exemple, en cryptant deux fois le mot de passe, on obtiendra un résultat crypté différent. Il s'agit donc ici de générer un mot de passe aléatoire et de l'envoyer par mail. La fonction d'envoi d'un mail étant créé par mon tuteur, je devais m'occuper de back lorsqu'il reçoit une demande de génération d'un nouveau mot de passe par un utilisateur.

```
const password = generator.generate({
    length: 10,
    numbers: true
})
```

Cette ligne de code me permet de générer un mot de passe aléatoire. Il me suffisait donc de le crypter et de la mettre dans la base de données pour ensuite le mettre dans le mail. Le texte dans un mail a la même structure qu'un message en HTML.

Ensuite, j'avais la charge de créer la fonctionnalité qui permet d'envoyer un mail vers l'utilisateur quand il s'inscrit qui doit contenir un lien pour la validation de son compte. Je devais créer le routage et utiliser la fonction pour construire le mail. Le lien de validation devait alors se diriger vers le back et contenir la clé qui permettrait de valider un compte.
Le lien aura la forme de « /api/users/validate?hash=XXXXX ». Pour pouvoir lire le paramètre hash dans l'URL, il suffit d'utiliser la ligne de code « req.query.hash ».
Il fallait ensuite savoir si cet « hash » était enregistrer dans la base de données et que si c'était le cas de valider le compte s'il n'était pas validé. L'utilisateur se fait rediriger vers le front avec un message si l'opération a réussi ou non.

Lorsque je devais ajouter ou modifier une table, il fallait aussi faire le changement dans le modèle du back. Par exemple, l'ajout de la table « bet_matchs » où il fallait définir l'interface (pour faciliter le typage des objets) et le modèle qui permet de configurer et d'indiquer au module comment la table est construite.

```
export const BetOnMatch = sequelize.define<BetOnMatchModel>('bet_matchs', {
    id: {
        autoIncrement: true,
        type: Sequelize.INTEGER,
    },
    userId: {
        field: 'user_id',
        type: Sequelize.INTEGER,
        primaryKey: true
    },
    matchId: {
        field: 'match_id',
        type: Sequelize.INTEGER,
```

Dans ce code, on peut voir qu'on définit les champs de la table en donnant leur type et leur contrainte. Par exemple pour le champ « match_id », en utilisant la clé « field », on indique que matchId se réfère à ce champ et permet d'utiliser matchId dans l'ensemble du code du back.

Le module « Sequelize » permet d'avoir une plus facile interaction avec la base de données. En effet, des méthodes sont déjà implémentés pour faire appel à la base. Par exemple, la méthode find() permet de faire un « select ». De ce fait, le développeur n'a pas besoin d'écrire de requête SQL. Il est toutefois possible d'utiliser des requêtes SQL pour interagir avec la base de données en utilisant la méthode query().

En plus du codage du front pour la page de profil, je devais m'occuper de sa partie back. Je devais récupérer les informations de l'utilisateur.

```
public async getMemberById(id: number): Promise<MemberModel | null> {
    return await Member.findOne({ where: { id: id } });
}
```

Toutes les méthodes dans les fichiers de repository sont asynchrones. findOne() permet de récupérer l'utilisateur en fonction de son identifiant. Ensuite vient la fonction de mise à jour des données :

Voici le code qui permet de mettre à jour le mot de passe :

```
const updated = await Member.update(
    { password: pass },
    { where: { id: userId } }
);
```

La méthode update() est directement implémentée dans le module « Sequelize ».

```
public async isPasswordOK(id: number, password: string): Promise<boolean> {
    const member = await Member.findOne({ where: { id: id } });
    if (member === null) {
        return Promise.resolve(false);
    }
    const isOk = await bcrypt.compare(password, member.password);
    if (isOk) {
        return Promise.resolve(true);
    }
    return Promise.resolve(false);
}
```

Le module bcrypt et sa fonction compare() permettent de comparer un code crypté et un code non crypté et de savoir si leur réel contenu est le même, ce qui est très utile dans un cryptage non bijectif.

Il est possible de mettre à jour la base données avec une API développée par un tier. Mon rôle consistait à créer un nouveau projet NodeJS qui avait pour but de faire des requête HTTP tous les débuts d'heure vers le back pour qu'il met à jour les données des équipes et des matchs. Pour cela il m'a fallu créer un nouveau projet NodeJS par la commande « npm init ». De me former sur les « crons » qui permettent de lancer des fonctions périodiquement selon la configuration. Voici le script du « cron » :

```
const CronJob = require('cron').CronJob;
const axios = require('axios');

const job = new CronJob('0 0 0-23 * * *', async() => {
    try {
        const target= `${process.env.API_URL}/api/livescore/standing`;
        const response = await axios.get(target);
        console.log(response);
    } catch(e) {

    }
}, null, false);
job.start();
```

Il a fallu installer les modules par les commandes suivantes :

```
npm install cron --save
```

Et

```
npm install axio --save
```

Axios permet de faire des requête HTTP.
Pour configurer la périodicité, il s'agit du champ encadré en rouge qui représente le temps de la forme : secondes minutes hours dayofMonth Month DayofWeek

Ici, cette fonction se lancera tous les débuts d'heure.

Le rôle du back est de faire le traitement afin que le front n'a pour but que d'afficher ce qu'il reçoit pour permettre d'avoir un site le plus rapide possible.

Par exemple, comme je l'ai expliqué dans la partie du front concernant la page « dashboard », il m'a fallu trouver dans le back le classement de l'utilisateur.

Avec l'aide de mon tuteur, j'ai pu écrire une requête SQL qui permet directement de récupérer le classement d'un joueur.

```
public async getOneRankMember(idUser: number) : Promise<any> {
    const q = `SELECT COUNT(*) FROM MEMBERS WHERE balance >
    (select balance from MEMBERS where id=${idUser} limit 1)`;
    const rank = await sequelize.query(q, { type: QueryTypes.SELECT } );
    return rank[0];
}
```

Il s'agit d'utiliser une sous-requête et COUNT qui permet d'avoir le nombre de joueur qui ont un meilleur score que l'utilisateur.

2.2.4. JWT : LES TOKENS

Les tokens d'authentification permet de gérer la connexion des utilisateurs sur le site web. A la différence avec les sessions, les tokens permettent aux deux serveurs (front et back) de savoir qui est connecté. Si des sessions sont établies dans le projet, le back ne pourra pas savoir si l'utilisateur est connecté ou non. Le module JWT pour Javascript Web Token permet de résoudre ce problème. Lorsque l'utilisateur se connecte au site, il fait la demande de la création d'un token qui dure 24 heures (selon la configuration choisie) et celui-ci doit être présenté en tête de requête pour que l'utilisateur puisse avoir accès aux fonctionnalités définies. Le token s'expire automatique après que le temps défini est dépassé et le front force la déconnexion de l'utilisateur.

Pour définir un chemin du routage dont le token est nécessaire, il suffit d'ajouter « auth » en paramètre du routage.

```
usersRouter.get("/getprofile", auth, async (req: Request, res: Response) => {
```

Par exemple, pour la page de profil où je souhaite récupérer les informations de l'utilisateur, définir « auth » dans le paramètre du routage permet aussi d'avoir l'identifiant de l'utilisateur dans la requête et est accessible de la manière suivante :

```
const id = req.body.userId;
```

L'une des tâches consistait à mettre à jour les informations de la session du front pour qu'il puisse savoir que les informations du back ont été changé. Modifier le

token n’est pas possible car un token ne doit pas être modifié pour garder son authenticité. Un token se présente ainsi :

```
eyJhbGciOiJIUzUxMiIsInR5cCI6IkpXVCJ9.eyJ
oZWxsbyI6Indvcmxk IiwibWVzc2FnZSI6IlRoYW5
rcyBmb3IgdmlzaXRpbmcgbm96emxlZ2Vhci5jb20
hIiwiaXNzdWVkIjoxNTU3MjU4ODc3NTI2fQ.NXd7
lC3rFLiNHXwefUu3OQ-R203pGfB87-dIrk2S-
vqfaygIWFwZKzmGHr6pzYkl2a0HkY0fdwa38yLWu
8Zdhg
```

La partie rouge est la tête du token, la partie violette représente le contenu du token qui est juste encodé et la partie bleue correspond au même contenu mais cryptées. Si le programme détecte une incohérence entre les informations encodées et cryptées, le token sera considéré comme obsolète.

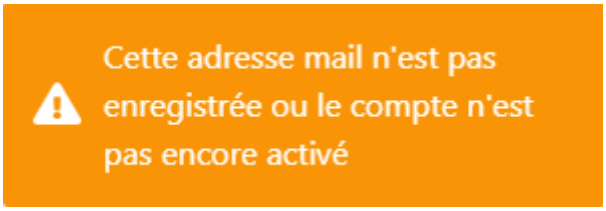
Ainsi pour faire part des changements sur le front, il a fallu changer le « local Storage », une variable accessible grâce à un service qui contient les informations de l’utilisateur.

2.2.5. LES INTERCEPTEURS

```
@Injectable()
export class ErrorInterceptor implements HttpInterceptor {
    constructor(private accountService: AccountService, private toastr: ToastrService) {

    intercept(request: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
        return next.handle(request).pipe(catchError(err => {

            if ([401, 403].includes(err.status) && this.accountService.userValue) {
                // auto logout if 401 or 403 response returned from api
                this.accountService.logout();
            }


            const error = err.error?.message || err.statusText;
            const code = err?.error?.code;

            switch (code) {
                case 'PASSWORD_NOT_MATCH':
                    this.toastr.warning('Votre mot de passe est incorrect');
                    break;
```

Le back peut envoyer vers le front des erreurs pour qu’il l’affiche à l’utilisateur. Ces erreurs apparaissent notamment sur la console.
Il est possible d’utiliser un middleware entre le back et le front qui permet d’intercepter les codes d’erreur grâce à HttpInterceptor pour afficher un message

d’alerte personnalisé. Grâce au module Toastr, on peut afficher des messages d’alerte selon le code erreur intercepté.
Par exemple si l’on intercepte le code erreur : ‘EMAIL_NOT_FOUND’, on affichera avec Toastr le message suivant :



Il existe plusieurs types de code de réponse du serveur par convention des développeurs :

- 200 : Pas de problème.
- 300 : Problème de la demande (erreur fonctionnelle)
- 400 : erreur coté client
- 500 : erreur coté serveur

2.2.6. AUTRE

Les tests unitaires n’ont pas été effectués durant ce projet par manque de temps car il s’agit d’un travail assez long et conséquent mais ils sont très importants de point de vue de compétences (peut être primordial pour un entretien technique lors d’un recrutement).

2.3. PROJET GESTION ANNUEL

Concernant ce projet, mon tuteur m'a laissé l'ensemble de la réalisation pour le début.

Après analyse de la problématique, j'ai conclu qu'il était plus pertinent de partir sur une base de données non relationnelle car avec un modèle relationnel, le nombre de tables seraient trop conséquents et NoSQL permet d'y remédier.

J'ai schématisé la base par un arborescence (voir annexe) et validé par mon tuteur.

Du fait que j'utilise du NoSQL, cela implique d'utiliser MongoDB.
On utilise donc pleinement le MEAN-stack.

La base de données utilise une collection qui contient les éléments du PDF.

Pour générer le PDF, il m'a fallu d'abord créer un formulaire qui récupère l'ensemble des informations de la fiche de mission.

Contrairement aux formulaires utilisés dans le projet de pronostics, ce projet utilise un autre module de création de formulaire qui est plus adapté pour les formulaires de grandes tailles et permet aussi de gérer les champs dynamiques. Ce module s'agit de FormGoup. Etant donné qu'il était construit de manière très différente à ngModel, il m'a fallu un certain temps pour comprendre et utiliser le module.

Voici le code qui permet d'initialiser le formulaire côté Typescript et qui permet de choisir les validations.

```
private _Form = this.fb.group({

  managers: this.fb.group({
    responsibles: this.fb.array([
      this.fb.control('', Validators.required)
    ]),
    redactor: this.fb.group({
      user: this.fb.control('', Validators.required),
      date: this.fb.control(new Date()),
      visa: this.fb.control('M.G')
    }),
    validator: this.fb.group({
      user: this.fb.control('', Validators.required),
      date: this.fb.control(new Date()),
      visa: this.fb.control('M.G')
    })

  }),

  provider: this.fb.group({
```

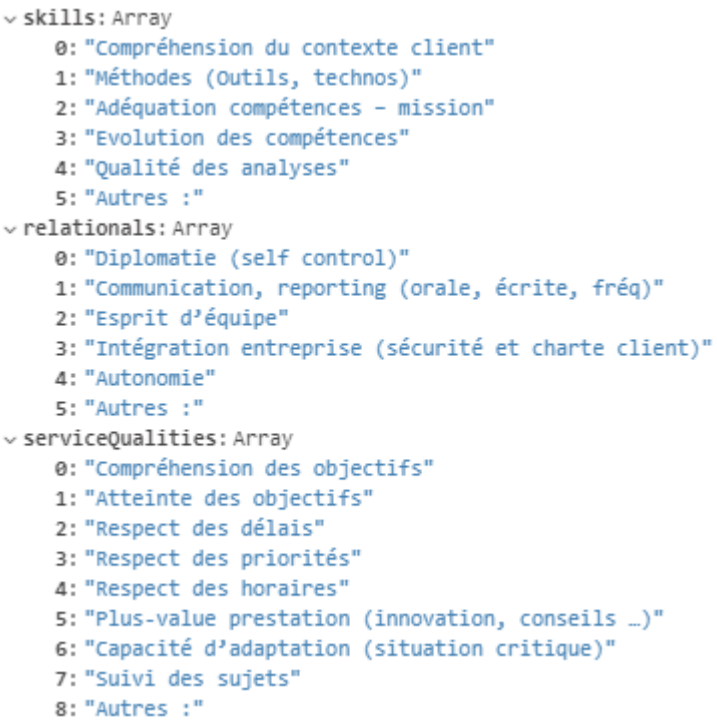
Pour établir ce code, il m'a fallu bien choisir et construire la façon dont la base de données est construite. Dans ce code, les « groups » sont équivalents à des objets, « array » à des listes et « control » à un champ du formulaire. « Validators » permet de choisir les contrôles sur le champ. Il était aussi possible d'établir une valeur par défaut sur le champ comme « new Date » ou « M.G ».

Le back est similaire à celui du site de pronostics, la seule différence étant l'utilisation de MongoDB à la place de PostgreSQL. Le modèle est donc différent et n'utilise non plus « Sequelize » mais « Mongoose ».

Il m'a fallu me former sur Mongoose, module qui permet de se connecter avec MongoDB. Pour cela, il faut définit des « Schemas » qui sont la représentation de la base de données dans le code du back. Les méthodes d'interactions avec la base de données sont similaires à ceux de Sequelize (find(), create()...).
Etant donné que les collections n'ont pas de structure réelle établie, il était possible de créer un script de validation qui permet de vérifier si un nouveau document répond bien aux contraintes établies (voir le script en annexe).
La difficulté se trouvait aussi dans la partie HTML où il fallait comprendre comment fonctionne les sous-objet notamment un objet de liste d'objet.

Ensuite, j'ai dû ajouter une collection pour les différents point mission pour les mettre dans la base de données au lieu de les mettre en code brut dans projet. De plus cela permet d'avoir une plus grande flexibilité (dans le cas où le PDF doit être changé).

Cette collection s'appelle « point » et a pour seul document :



MongoDB et Mongoose introduisent la notion de « ObjectId » qui fait le lien entre les documents. ObjectId est un nouveau type instauré par Mongoose dans le back. « ObjectId » est considéré comme un objet bien que lorsque l'on veut l'afficher, il apparait comme une chaîne de caractère. Il s'agit en réalité d'une chaîne en hexadécimal et peut être obtenu en convertissant une chaîne de caractère compatible. Cet « ObjectId » est unique à chaque document (similaire à la clé primaire dans une table) et généré automatiquement par la base de données.

Enfin, pour la mise en forme de la page, j'ai installé et utilisé Bootstrap qui permet d'avoir un résultat satisfaisant rapidement. En effet, l'installation est très rapide grâce à NPM par la commande : « npm install bootstrap --save »

Concernant la génération de PDF à partir des données de la base, j'ai choisi et essayé le module « pdfmake » (installable avec la commande : « npm install pdfmake --save »). Le but est de générer au back le fichier PDF et l'envoyer au front pour que l'utilisateur puisse la télécharger. Le module était cependant assez difficile d'utilisation et par manque de temps (fin du stage), la fonctionnalité n'a pas abouti à un résultat. Pour la suite du projet, c'est mon tuteur qui devait le continuer.

Conclusion

Durant ce stage, j'ai pu découvrir l'environnement professionnel et le monde des développeurs. J'ai pu apprendre à utiliser les outils qui permettent de gérer les projets.

J'ai appris à utiliser les technologies annoncées durant ce rapport et appris aussi à m'adapter et avoir un esprit critique sur les technologies que l'on pourrait utiliser pour prendre le plus pertinent selon la problématique et la solution souhaitée.

En termes connaissances de développeur, j'ai pu apprendre à utiliser les technologies Full-stack de Javascript qui sont des technologies très dynamiques de part sa communauté de développeurs qui l'améliorent de part la création de modules ou de librairies.

ANNEXES

LIEN DU PROJET INSPIRE POUR LA BOITE DE DIALOGUE EN ANGULAR

https://stackblitz.com/edit/angular-confirmation-dialog

SCRIPT TRIGGER UTILISEE DANS LA BASE DE DONNES DU SITE DE PRONOSTIC

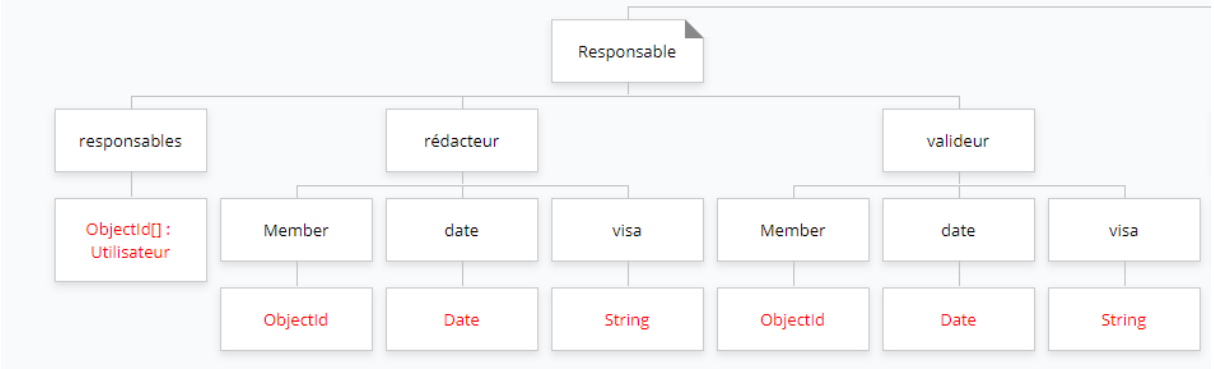
```
CREATE FUNCTION sum_score_bet_teams(_id bigint)

returns integer
LANGUAGE 'sql'

as $$
SELECT COALESCE(SUM(score),0) as _sum
FROM bet_teams
WHERE is_done='true' AND is_won='true' AND user_id=_id;
$$;




CREATE FUNCTION sum_score_bet_matchs(_id bigint)
returns integer
LANGUAGE 'sql'
as $$
SELECT COALESCE(SUM(score),0) as _count
FROM bet_matchs
WHERE is_done='true' AND is_ok='true' AND user_id=_id;
$$;




CREATE FUNCTION count_total_bet_teams(_id bigint)

returns integer
LANGUAGE 'sql'

as $$
SELECT COALESCE(COUNT(*),0) as _count
FROM bet_teams
WHERE user_id=_id;
$$;


CREATE FUNCTION count_total_bet_matchs(_id bigint)

returns integer
LANGUAGE 'sql'
```

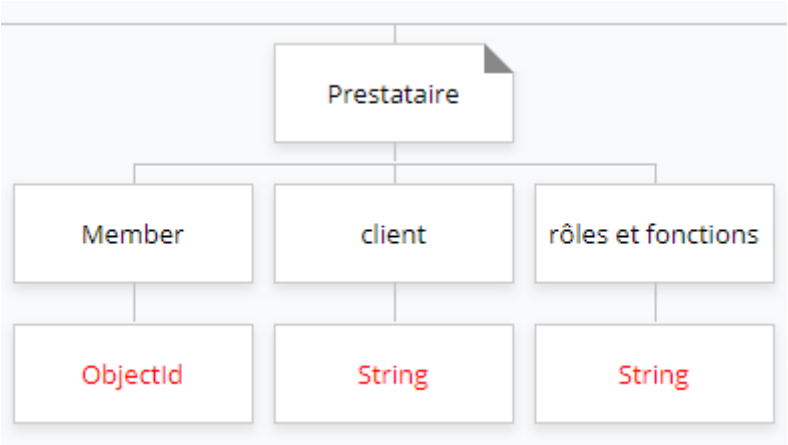
```
as $$
SELECT COALESCE(COUNT(*),0) as _count
FROM bet_matchs
WHERE user_id=_id;
$$;



CREATE FUNCTION count_total_successful_bet_teams(_id bigint)

returns integer
LANGUAGE 'sql'

as $$
SELECT COALESCE(COUNT(*),0) as _count
FROM bet_teams
WHERE is_done='true' AND is_won='true' AND user_id=_id;
$$;



CREATE FUNCTION count_total_successful_bet_matchs(_id bigint)

returns integer
LANGUAGE 'sql'

as $$
SELECT COALESCE(COUNT(*),0) as _count
FROM bet_matchs
WHERE is_done='true' AND is_ok='true' AND user_id=_id;
$$;



CREATE FUNCTION update_member_bet_teams() returns TRIGGER
LANGUAGE PLPGSQL
as $$
BEGIN

IF TG_OP = 'DELETE' THEN UPDATE members SET balance =
sum_score_bet_teams(OLD."user_id") + sum_score_bet_matchs(OLD."user_id"),
total_bets_teams = count_total_bet_teams(OLD."user_id")
WHERE id=OLD."user_id";

ELSIF TG_OP = 'UPDATE' THEN
```

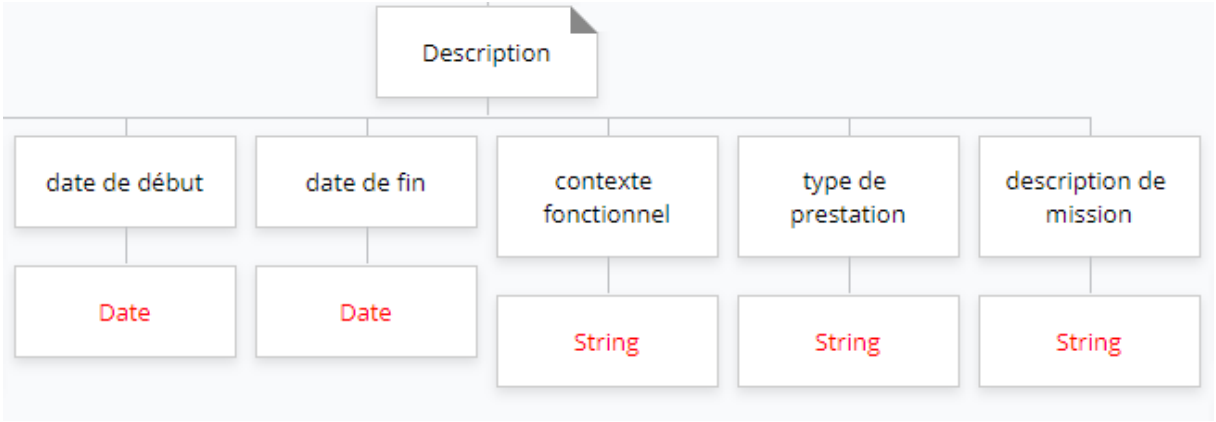
```
UPDATE members SET balance = sum_score_bet_teams(NEW."user_id") +
sum_score_bet_matchs(NEW."user_id") , total_successful_bets_teams =
count_total_successful_bet_teams(NEW."user_id")
WHERE id=NEW."user_id";


ELSIF TG_OP = 'INSERT' THEN
UPDATE members SET balance = sum_score_bet_teams(NEW."user_id") +
sum_score_bet_matchs(NEW."user_id"), total_bets_teams =
count_total_bet_teams(NEW."user_id")
WHERE id=NEW."user_id";


END IF;

RETURN NULL;

END;

$$;


CREATE FUNCTION update_member_bet_matchs() returns TRIGGER
LANGUAGE PLPGSQL
as $$
BEGIN

IF TG_OP = 'DELETE' THEN
UPDATE members SET balance = sum_score_bet_teams(OLD."user_id") +
sum_score_bet_matchs(OLD."user_id"), total_bets_matchs =
count_total_bet_matchs(OLD."user_id")
WHERE id=OLD."user_id";


ELSIF TG_OP = 'UPDATE' THEN
UPDATE members SET balance = sum_score_bet_teams(NEW."user_id") +
sum_score_bet_matchs(NEW."user_id"), total_successful_bets_matchs =
count_total_successful_bet_matchs(NEW."user_id")
WHERE id=NEW."user_id";


ELSIF TG_OP = 'INSERT' THEN
UPDATE members SET balance = sum_score_bet_teams(NEW."user_id") +
sum_score_bet_matchs(NEW."user_id"), total_bets_matchs =
count_total_bet_matchs(NEW."user_id")
WHERE id=NEW."user_id";

END IF;

RETURN NULL;
```

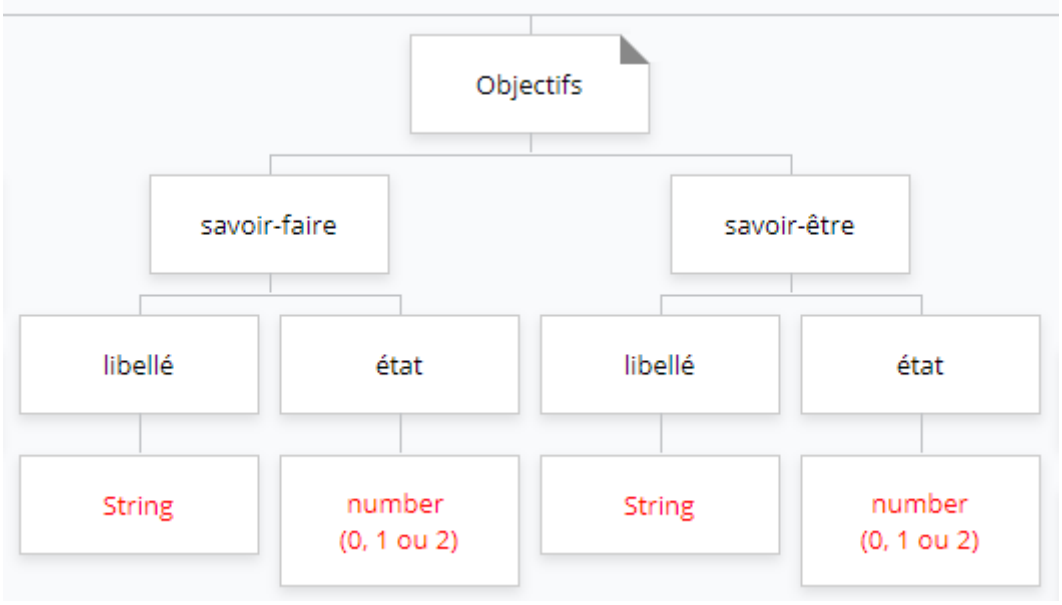
```
END;
$$;


CREATE TRIGGER after_insert_update_delete_bet_teams
AFTER INSERT OR UPDATE OR DELETE ON bet_teams
FOR EACH ROW
EXECUTE PROCEDURE update_member_bet_teams();

CREATE TRIGGER after_insert_update_delete_bet_matchs
AFTER INSERT OR UPDATE OR DELETE ON bet_matchs
FOR EACH ROW
EXECUTE PROCEDURE update_member_bet_matchs();
```

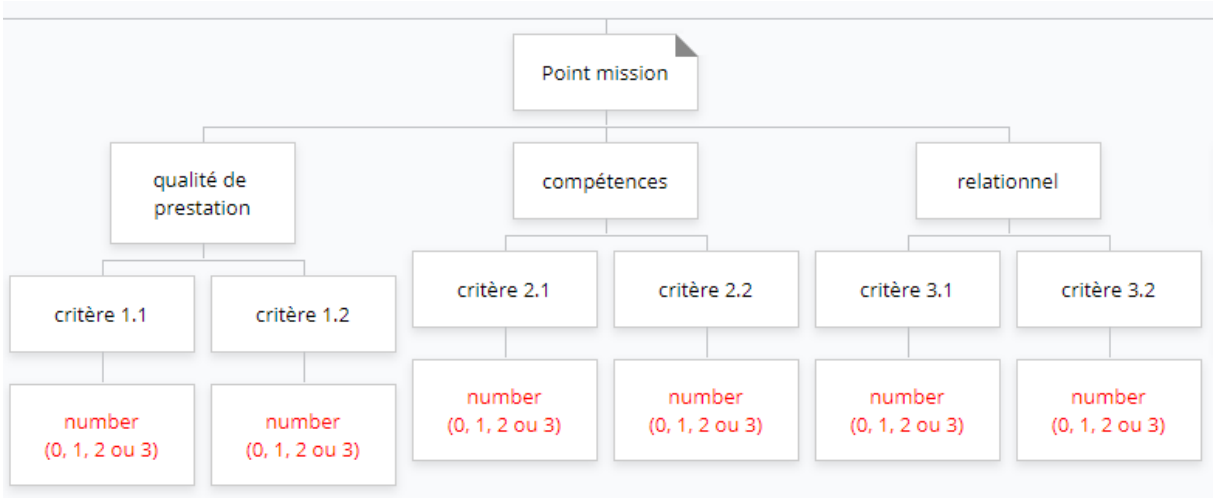
ARBORESSANCE DE LA BASE DE DONNEES COLLECTION MISSION DU PROJET GESTION ANNUELLE

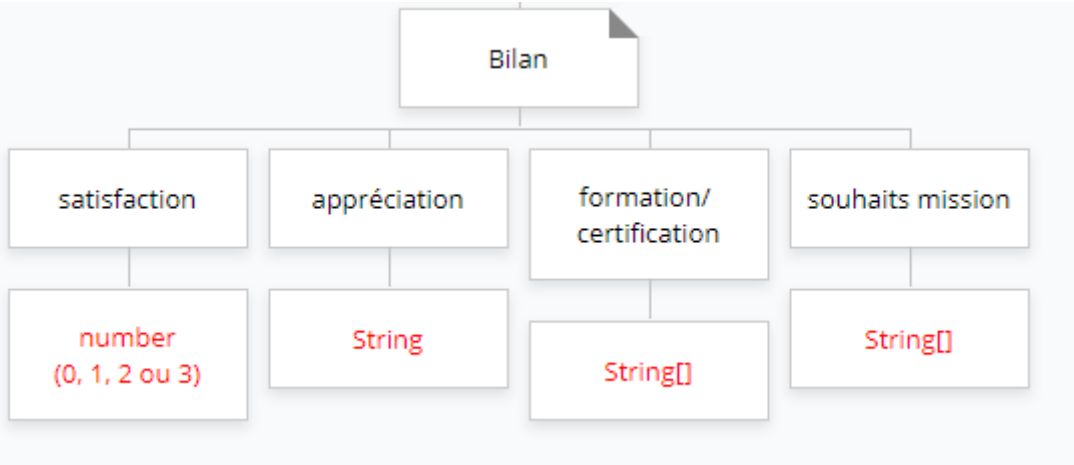





Description
date de début
date de fin
contexte fonctionnel
type de prestation
description de mission
Date
Date
String
String
String


Objectifs
savoir-faire
savoir-être
libellé
état
libellé
état
String
number (0, 1 ou 2)
String
number (0, 1 ou 2)


Point mission
qualité de prestation
compétences
relationnel
critère 1.1
critère 1.2
critère 2.1
critère 2.2
critère 3.1
critère 3.2
number (0, 1, 2 ou 3)
number (0, 1, 2 ou 3)
number (0, 1, 2 ou 3)
number (0, 1, 2 ou 3)
number (0, 1, 2 ou 3)
number (0, 1, 2 ou 3)



SCRIPT DE VALIDATION COLLECTION

```
{
  $jsonSchema: {
    bsonType: 'object',
    required: [
      'managers',
      'provider',
      'description',
      'objectives',
      'point',
      'results'
    ],
    properties: {
      managers: {
        bsonType: 'object',
        description: 'must be object',
        required: [
          'redactor',
          'validator',
          'responsibles'
        ],
        properties: {
          redactor: {
            bsonType: 'object',
            description: 'must be object',
            required: [
              'user',
              'date',
              'visa'
            ],
            properties: {
              user: {
```

```
            bsonType: 'string',
            description: 'must be objectId'
          },
          date: {
            bsonType: 'date',
            description: 'must be date'
          },
          visa: {
            bsonType: 'string',
            description: 'must be string'
          }
        }
      },
      validor: {
        bsonType: 'object',
        description: 'must be object',
        required: [
          'user',
          'date',
          'visa'
        ],
        properties: {
          user: {
            bsonType: 'string',
            description: 'must be objectId'
          },
          date: {
            bsonType: 'date',
            description: 'must be date'
          },
          visa: {
            bsonType: 'string',
            description: 'must be string'
          }
        }
      },
      responsibles: {
        bsonType: [
          'array'
        ],
        description: 'must be array',
        minItems: 1,
        items: {
          bsonType: 'string',
          description: 'must be objectId'
        }
      }
    }
  },
  provider: {
```

```
        bsonType: 'object',
        description: 'must be object',
        required: [
          'user',
          'client',
          'roles_functions'
        ],
        properties: {
          user: {
            bsonType: 'string',
            description: 'must be objectId'
          },
          client: {
            bsonType: 'string',
            description: 'must be string'
          },
          roles_functions: {
            bsonType: 'string',
            description: 'must be string'
          }
        }
      },
      description: {
        bsonType: 'object',
        description: 'must be object',
        required: [
          'technical_environment',
          'starting_date',
          'ending_date',
          'fonctional_context',
          'service_type',
          'mission_description'
        ],
        properties: {
          technical_environment: {
            bsonType: 'string',
            description: 'must be string'
          },
          starting_date: {
            bsonType: 'date',
            description: 'must be date'
          },
          ending_date: {
            bsonType: 'date',
            description: 'must be date'
          },
          functional_context: {
            bsonType: 'string',
            description: 'must be string'
          },
```

```
        service_type: {
          bsonType: 'string',
          description: 'must be string'
        },
        description_mission: {
          bsonType: 'string',
          description: 'must be string'
        }
      }
    },
    objectives: {
      required: [
        'skills',
        'attitudes'
      ],
      properties: {
        skills: {
          bsonType: [
            'array'
          ],
          description: 'must be array',
          items: {
            bsonType: 'object',
            description: 'must be object',
            required: [
              'wording',
              'condition'
            ],
            properties: {
              wording: {
                bsonType: 'string',
                description: 'must be string'
              },
              condition: {
                'enum': [
                  'Non Atteint',
                  'En cours',
                  'Atteint'
                ]
              }
            }
          }
        },
        attitudes: {
          bsonType: [
            'array'
          ],
          description: 'must be array',
          items: {
            bsonType: 'object',
```

```
            description: 'must be object',
            required: [
              'wording',
              'condition'
            ],
            properties: {
              wording: {
                bsonType: 'string',
                description: 'must be string'
              },
              condition: {
                'enum': [
                  'Non Atteint',
                  'En cours',
                  'Atteint'
                ]
              }
            }
          }
        }
      }
    },
    point: {
      bsonType: 'object',
      description: 'must be object',
      required: [
        'service_quality',
        'skills',
        'relational'
      ],
      properties: {
        service_quality: {
          bsonType: [
            'array'
          ],
          description: 'must be array',
          items: {
            bsonType: 'object',
            description: 'must be object',
            required: [
              'wording',
              'level'
            ],
            properties: {
              wording: {
                bsonType: 'string',
                description: 'must be string'
              },
              level: {
                'enum': [
```

```
            0,
            1,
            2,
            3
          ]
        }
      }
    }
  },
  skills: {
    bsonType: [
      'array'
    ],
    description: 'must be array',
    items: {
      bsonType: 'object',
      description: 'must be object',
      required: [
        'wording',
        'level'
      ],
      properties: {
        wording: {
          bsonType: 'string',
          description: 'must be string'
        },
        level: {
          'enum': [
            0,
            1,
            2,
            3
          ]
        }
      }
    }
  },
  relational: {
    bsonType: [
      'array'
    ],
    description: 'must be array',
    items: {
      bsonType: 'object',
      description: 'must be object',
      required: [
        'wording',
        'level'
      ],
      properties: {
```

```
            wording: {
              bsonType: 'string',
              description: 'must be string'
            },
            level: {
              'enum': [
                0,
                1,
                2,
                3
              ]
            }
          }
        }
      }
    }
  },
  results: {
    bsonType: 'object',
    description: 'must be object',
    required: [
      'satisfaction',
      'appreciation',
      'training_certification',
      'mission_wishes'
    ],
    properties: {
      satisfaction: {
        'enum': [
          0,
          1,
          2,
          3
        ]
      },
      appreciation: {
        bsonType: 'string',
        description: 'must be string'
      },
      training_certification: {
        bsonType: [
          'array'
        ],
        description: 'must be array',
        items: {
          bsonType: 'string',
          description: 'must be string'
        }
      },
      mission_wishes: {
```

```
            bsonType: [
              'array'
            ],
            description: 'must be array',
            items: {
              bsonType: 'string',
              description: 'must be string'
            }
          }
        }
      }
    }
  }
}
```