Composition du groupe

William MADIE
Pascal DU
Louis BIBAL
Katia DRICI
Linda BESSAH
Kévin TRUONG
Emma CHEN
Yewon KANG
Tefy ANDRIANIRAISANTSOA

Choix architecturaux

Pour construire notre API, nous avons opté pour le framework **Spring Boot**. Nous avons fait ce choix afin de gagner du temps dans notre développement (installation environnement de développement, Gestion des dépendances facilitée, etc). Ce framework impose également une structure de projet, il est dit opinionated. Cela incite à appliquer les principes SOLID lors du développement, ce qui facilite la collaboration et l'extension. Par la suite, nous avons choisi une structure de projet dans laquelle tous les éléments sont groupés par concepts. Ainsi, les controllers, services, entités, DTOs, etc sont groupés dans un même dossier. Cela permet d'ajouter de nombreuses fonctionnalités tout en permettant un accès rapide à tous les éléments impliqués dans une fonctionnalité.

Emma, Kévin, Pascal

Sur les deux fonctionnalités, nous avons repris la même cohérence des dossiers, pour expliquer chaque dossier brièvement, nous avons un dossier **Persistence** qui permet de faire le lien avec la base de données, puis un dossier **Controller** qui gère les flux (CRUD) et finalement un dossier **Service** qui gère les fonctionnalités métiers.

Nous avons implémenté la fonctionnalité **Profil des compétiteurs**. Nous avons décidé d'importer dans notre base de donnée un fichier csv que nous avons traité pour obtenir uniquement les informations utiles et traitables pour notre fonctionnalité. Pour ce qui est des requêtes à la base de données, nous avons utilisé un service **AthleteService** qui permet de récupérer les données de la base de données, nous avons fait ce choix pour respecter le principe SOLID, notamment pour respecter le principe de ségrégation des interfaces.

Nous avons également implémenté la fonctionnalité Météo, pour cela nous avons utilisé une API (openweathermap.org) qui nous retournait en format json les prévisions d'une ville, nous avons stocké l'URL et le Token dans le fichier **.properties** permettant de stocker la configuration. On a également manipulé la base de données avec le fichier **MeteoService** pour récupérer toutes les villes candidates aux JO, pour retourner au client les prévisions de chaque ville.

Katia, Linda, William

Nous avons développé la fonctionnalité **Boutique d'articles exclusifs**. Nous l'avons un peu remanié de manière à pouvoir gérer des accessoires et des vêtements. Nous avons utilisé une interface (shopService) qui doit être implémentée par tous les nouveaux services de la boutique. Cela permet une extension facile respectant les principes SOLID. Cette fonctionnalité ne communique pas avec d'autres fonctionnalités (ni externes, ni internes). Cependant, elle est complète du point de vue CRUD. Il est possible d'ajouter un article, récupérer le détail de l'article, voir tous les articles connus et enfin, diminuer la quantité d'un article connu).

Nous avons mis en place une BDD de développement commune accessible à tous les développeurs du projet. Nous avons aussi configuré l'accès à la BDD depuis l'API (dépendances). Enfin, nous avons géré la configuration initiale de l'API afin que tout le monde puisse partir sur une base commune (README.md avec les informations de connexion à la BDD, Documentation Swagger).

Yewon, Louis, Tefy

Nous avons développé la fonctionnalité **Informations sur les pays participants**. Nous avons utilisé l'API REST Countries qui nous retournait certaines informations à propos de chacun des pays du monde. Les informations retournées étaient, pour chaque pays, son nom, sa capitale, sa région (Europe, Asie, etc.), sa ou ses langue(s), un lien Google Maps pour visualiser le pays, son nombre d'habitants ainsi qu'une image de son drapeau. Nous considérions uniquement les pays indépendants. Sur cette fonctionnalité, nous avons repris la même cohérence des dossiers que les autres membres du projet, à savoir un dossier **Persistence** contenant la classe Country modélisant chacun des pays sous la forme d'objets avec des attributs. Nous avions, également, un dossier **Controller** contenant la classe CountryController à laquelle l'on effectue des requêtes afin qu'elle nous renvoie les informations pour un pays ou pour tous les pays. Enfin, on dispose également d'un dossier **Service** qui gère les fonctionnalités métiers avec l'API. Ce dossier est constitué d'une interface **CountryDataProvider** qui permet de garantir que si l'on change de procédé de récupération des données, on aura toujours les mêmes services réalisables. Ce dossier comprend également une classe **APICountryDataProvider** qui implémente l'interface précédente et qui va utiliser la classe **APIClient**. Cette dernière va envoyer des requêtes à l'API. Elle récupère alors les informations souhaitées au format JSON et les transforme en objets Country afin de les transmettre à la classe APICountryDataProvider puis au contrôleur. Dans la base de données du projet, nous disposions d'une table Pays associant le nom d'un pays participant à son code à deux lettres (FR, etc). C'est ce dernier code qui peut être transmis à l'API afin d'obtenir les informations pour un pays bien spécifique.

Pour la fonctionnalité Statistiques sportives, nous avons créé une table SQL appelée "categorie" qui catégorise chaque sport en 3 types. Nous avons également créé une autre table appelée "statistic" pour enregistrer les statistiques de chaque athlète. Cependant, les attributs des statistiques diffèrent en fonction du sport (par exemple, les sports collectifs ont un attribut "nbPasseDecisive" tandis que les sports individuels ont un attribut "classmentMondial"). Au lieu de stocker des nombres directs dans les colonnes, nous stockons des fichiers JSON dans les données

L'interface **PerformanceCreator** contient une fonction create. Chaque classe du catagorie du sport hérite de cette interface et crée une statistique en fonction de sa catégorie de sport. La classe **PerformanceFactory** dirige vers la classe appropriée en fonction de la catégorie du sport. Ensuite, les chiffres générés sont sauvegardés dans la base de données en utilisant la classe **StatisticRepository**. La classe StatisticRepository comprend également une fonction telle que getStatisticOfAthlete qui permet de récupérer les statistiques d'un athlète. La classe **StatisticController** comprend deux fonctions : comparerAthletes et obtenirInformations. La fonction CompareAthelet permet à l'utilisateur de spécifier deux noms d'athlètes. La fonction getInformations permet à l'utilisateur de spécifier un tableau de noms d'athlètes. Les statistiques de chaque athlète sont récupérées à partir du StatisticRepository et affichées simplement.