

ORION



Justification des choix techniques ***Projet MDD***



Auteur : Manon Antigone Dauguet
Version 0.0.1

Aperçu / Synthèse	3
Choix techniques généraux.....	4
Architecture	4
Type d'API	4
Sécurité front-back	5
Token JWT	5
Maintien de connexion sécurisé	5
Choix techniques côté back.....	6
Langage et framework	6
Spring Boot.....	6
Modules et starters clefs	6
Librairies de test	7
Autres dépendances	8
La base de données	8
Spring Data et l'aspect relationnel des tables	8
Données mockées	9
Petit point sur les endpoints	9
Choix techniques côté front	11
Langage et framework	11
Librairies de test	11
Autres dépendances	11
Observables et économie de requêtes	12

Aperçu / Synthèse

Les choix techniques réalisés sont mis au service de la robustesse, de la performance, de l'évolutivité et de l'abaissement des coûts de production de l'application.

Les langages choisis sont java (framework Spring Boot) et Angular. L'architecture est orientée service, et le côté serveur héberge une base de données ainsi qu'une API RESTful par laquelle le côté client interagit avec le serveur.

Pour assurer une connexion sécurisée, chaque requête http (autre que celles liées à l'authentification) est sécurisée par un token JWT unique et personnel à l'utilisateur. Ce token est fourni par le back au front via un cookie, et ajouté à chaque requête émise via un `HttpInterceptor`. Ceci permet un maintien sécurisé et efficace de la connexion entre deux sessions. De plus, les requêtes http redondantes sont évitées en stockant l'état des données utilisées côté client via des `BehaviorSubject`.

Des données mockées ont été ajoutées à la base de données pour faciliter le test de l'application.

Dans un premier temps, seuls les services, côté back et front, se sont vus ajouter des tests automatisés. Les bibliothèques JUNIT5, Mockito et Jest ont été mobilisées pour faciliter la mise en place de tests robustes.

Choix techniques généraux

Architecture

choix technique	lien vers le site / la documentation / une ressource	but du choix
Architecture orientée service	https://www.ibm.com/docs/en/was/8.5.5?topic=ws-service-oriented-architecture	Mettre en place une architecture propice à la lisibilité, la maintenance, la performance et l'évolutivité du projet.

Une **architecture orientée service** bénéficie des avantages d'une architecture modulaire et en couche : chaque service est un bloc indépendant et réutilisable, organisé en couches pour séparer la logique métier, l'accès aux données et l'affichage.

De plus, son grand intérêt réside dans un découpage en services indépendants, responsables de fonctionnalités spécifiques (gestion des utilisateurs, des pots...), interagissant entre eux par l'intermédiaire d'API. Ainsi, l'indisponibilité d'un service n'empêche pas les autres de fonctionner et toutes les sollicitations ne concernent pas le même composant (potentiellement rapidement surchargé).

De part leur indépendance, l'ajout ou la modification d'un service est également facilité, améliorant la scalabilité du projet.

Type d'API

choix technique	lien vers le site / la documentation / une ressource	but du choix
API RESTful	https://restfulapi.net/	Garantir des interactions efficaces, standardisées et facilement maintenables entre le front et back.

Le front de notre application communiquera avec le back par l'intermédiaire d'une API suivant les principes **REST**. C'est un modèle très utilisé, performant et très documenté.

Parmi les normes de ce modèle, on retrouve la séparation client-serveur (communiquant ensemble par le biais de protocoles http), le stateless (requêtes indépendantes), le cacheable (sauvegardable en local), l'uniformité (interfaces compatibles entre plusieurs API), le système de couche (composants

indépendants), et le code à la demande (optionnel, possibilité de fournir du code exécutable au client, comme des scripts, pour étendre ses capacités).

Sécurité front-back

choix technique	lien vers le site / la documentation / une ressource	but du choix
JSON Web Token	https://jwt.io/introduction	Sécuriser les interactions entre le front et le back tout en respectant l'aspect stateless du RESTful.
HttpInterceptor	https://angular.fr/http/interceptor	Limiter la redondance de code lors de l'ajout du token aux requêtes, et centraliser la gestion de la sécurité côté front

Token JWT

Les **tokens JWT** (JSON Web Tokens) sont une solution légère, moderne et très efficace pour sécuriser l'authentification des utilisateurs. Une authentification réussie, via l'email et le mot de passe envoyés par l'utilisateur au moment du login, fournit une chaîne de caractère encodée et personnelle qui sera fournie à chaque requête.

Chaque requête passe par une vérification de la validité de ce token avant de s'exécuter et de répondre (respectant l'aspect stateless de l'API). Il s'agit d'une vérification forte de l'identité de l'utilisateur qui évite de transiter des données sensibles (comme le mot de passe) à chaque requête.

Maintien de connexion sécurisé

Le maintien de la connexion entre deux sessions va reposer sur la sauvegarde du token.

Côté back, le token est envoyé sous forme de **cookie « HttpOnly »**, empêchant son accès via JavaScript et réduisant les risques de vol par attaque XSS. Ce choix permet également d'éviter le stockage du token dans le localStorage, limitant ainsi les risques de fuite et d'exploitation malveillante.

Côté front, une implémentation de **HttpInterceptor** sera responsable de l'ajout du token (stocké dans le cookie) à chaque requête émise.

Choix techniques côté back

Langage et framework

choix technique	lien vers le site / la documentation / une ressource	but du choix
Java Spring Boot	https://spring.io/projects/spring-boot	Créer à moindre coût une application robuste, évolutive et sécurisée.
Modules clefs		
Spring MVC	https://docs.spring.io/spring-framework/reference/web/webmvc.html	Faciliter la gestion des requêtes http.
Spring Data	https://spring.io/projects/spring-data	Faciliter la mise en place et la gestion de la base de données. Faciliter les interactions sécurisées avec la base de données.
Spring Security	https://spring.io/projects/spring-security	Mettre en place une sécurisation efficace des interactions entre le front et le back.

Spring Boot

Java est un langage robuste et performant. Son typage fort et ses nombreuses librairies offrent une base fiable pour une API sécurisée, performante et évolutive. Du fait qu'il soit adopté massivement en entreprise, sa documentation est riche, sa communauté importante, et support relativement assuré à long terme.

Son framework **Spring Boot**, largement éprouvé, permet un gros gain sur le coût de développement en fournissant des outils simplifiant la mise en place de l'API. De plus, il apportera également des fonctionnalités clefs pour la sécurité de l'application.

Modules et starters clefs

spring-boot-starter-validation nous fournit les annotations telles que @Valid ou @NotNull pour faciliter le contrôle des éléments envoyé à l'API, comme la création d'un nouvel utilisateur par exemple avec un email et un mot de passe valides.

Spring MVC permet de gérer les requêtes HTTP de manière claire et structurée, en utilisant des Controllers pour l'implémentation des différentes actions de l'API. Grâce à ses annotations telles que `@RestController` ou `@GetMapping`, il facilite la gestion des endpoints de l'application.

Spring Data simplifie l'accès aux données. Il apporte notamment les « JpaRepository », permettent l'utilisation de méthodes pré-implémentées pour interagir avec la base de données, et la création de requêtes personnalisée via des méthodes nommées (telle « `findByOrderByCreatedAtDesc` »). Spring Data permet ainsi de « cacher » une partie de la logique SQL, réduisant les risques d'erreurs dans les requêtes, tout en optimisant les requêtes effectuées. Spring Data facilite également la relation entre les entités, point que nous détaillerons dans la partie « La base de données ».

Spring Security fournit des outils pour intégrer une gestion robuste et efficace de la sécurité de l'application. Il permet d'intégrer l'authentification des utilisateurs. Dans notre cas, cette authentification est basée sur les token JWT définis en partie « Choix de sécurité front-back ».

Librairies de test

choix technique	lien vers le site / la documentation / une ressource	but du choix
JUnit5	https://junit.org/junit5/	Facilite le développement de tests robustes.
Mockito	https://site.mockito.org/	Simule les dépendances du code testé.

L'utilisation de ces librairies fera gagner l'application en robustesse (assurant davantage sa disponibilité et son évolutivité), à moindre coût d'effort et de temps.

Mockito est intégré pour isoler et simuler les dépendances externes dans les tests, permettant de vérifier chaque composant de manière indépendante. Mocker les données sera nécessaire pour une application robuste, et Mockito a l'avantage d'être très utilisé et facile d'utilisation.

JUnit5 est le framework de test standard pour les applications Java. En plus d'être documenté et éprouvé, ce framework apporte de puissantes améliorations sur la rapidité d'implémentation, la lisibilité et le paramétrage des tests. JUnit5 a aussi l'avantage de très bien fonctionner avec Spring Boot et Mockito.

Autres dépendances

choix technique	lien vers le site / la documentation / une ressource	but du choix
JWT	https://github.com/jwtk/jjwt	Facilite la création et la validation des token JWT.
Springdoc-openapi	https://springdoc.org/	Fournit une documentation SWAGGER à l'API.
Lombok	https://projectlombok.org/	Facilite la gestion des données.
Dotenv-java	https://github.com/cdimascio/dotenv-java	Offre une façon sécurisée et simple de se connecter à la base de données pour le test du MVP.
Mysql-connector-j	https://github.com/mysql/mysql-connector-j	Indispensable pour communiquer avec notre base de données.

JWT est une library simple d'utilisation, très populaire et très documentée. Elle permet de générer, signer et valider des tokens JWT sans dépendances externes.

Springdoc-openapi génère automatiquement une documentation SWAGGER de l'API, fournissant une source fiable et à jour pour l'utilisation de nos différents endpoints.

Lombok permet de réduire la quantité de code répétitifs en générant les getters, setters et autres méthodes courantes. C'est un gain de temps pour un code plus lisible.

Dotenv-java est une solution temporaire. Il permet de stocker les identifiants de la base de données de façon sécurisée dans un fichier .env à part, sollicité par notre application.properties. En production, cette solution n'aura plus lieu d'être car il sera alors plus simple et sécurisé d'entrer les identifiants et autres paramètres dans les variables d'environnement ou en ligne de commande.

Mysql-connector-j est un driver JDBC (Java Database Connectivity). Il permet à notre application en java de se connecter à notre base de données en MySQL.

La base de données

Spring Data et l'aspect relationnel des tables

Les connexions entre les différentes tables sont reléguées à Spring Data et ses annotations @ManyToOne, @ManyToMany ou encore @JoinColumn.

Ma première idée était un stockage des id, dans des tableaux, pour connecter les tables entre elles. Par exemple, une entité Post qui aurait comme attribut « comments » un tableau contenant les id des différents commentaires concernant l'article. Ceci posait un problème d'optimisation à 2 niveaux :

- D'une part, certaines requêtes devraient s'écrire manuellement, nous privant des avantages de Spring Data. Par exemple, récupérer tous les commentaires d'un article.
- D'autre part, à chaque mise à jour d'une entité, il faut s'assurer manuellement d'une mise à jour correcte des tableaux stockant son id. Ceci alourdit grandement les requêtes d'ajout ou de suppression de données.

Avec Spring Data, les annotations prennent en charge les relations entre les entités, sans requêtes sql pour réaliser les jointures (« SELECT c FROM Comment c WHERE c.id IN :commentIds.. »). Cela permet d'exploiter au mieux les méthodes prédéfinies d'interaction avec la base de données dont nous avons déjà parlé. De plus, les mises à jour des données sont automatiquement étendues à toutes les tables directement ou indirectement concernées.

Données mockées

Avant la phase de production, les configurations entourant la base de données ont été pensées pour faciliter le test de l'application dans son format minimal.

Ainsi, des données mockées, avec quelques utilisateurs, thèmes, articles et commentaires, sont intégrés à la base de données. A chaque run de l'application, la base de données est entièrement rebootée.

Une configuration similaire est mise en place pour l'environnement de test, mais cette configuration n'a pas vocation à changer à terme.

Petit point sur les endpoints

Les endpoints ont été pensés pour être intuitifs et propices à une complexification future de l'application.

Nous sommes en droit de supposer, qu'à terme, les articles et les commentaires sur chaque article seront très nombreux. Trop pour être récupérés en une seule fois, sans compromettre grandement la performance et la rapidité de l'application. Les endpoints ont donc été pensés pour qu'à terme un système de pagination visant à charger les 50 premiers résultats soit possible.

- Récupération des commentaires d'un post

La liste des commentaires d'un article n'est pas intégrée à l'objet rendu par l'API

lors de la récupération d'un article (requête « **post/{Xid}** »). Elle est rendue après une requête « **post/{Xid}/comment** ». A terme, la requête chargera les 50 premiers commentaires, puis éventuellement les 50 suivants.

Il est beaucoup intuitif de poster un nouveau commentaire sur une route « **post/{Xid}/comment** ». A terme, une route telle que « **post/{Xid}/comment/{comment id}** » pourrait exister pour mettre à jour ou supprimer un commentaire spécifique.

- Récupération des tous les posts auquel un utilisateur est abonné

Un problème de performance c'est aussi posé lors de la récupération de tous les posts, non pas d'un topic, mais de tous les topics auquel l'utilisateur est abonné. La logique reste permettre à terme de ne charger que les premiers résultats d'une lourde requête. Une route « **post/subscriber** » est donc mise en place, plutôt qu'une route « **topic/{id}/post** », répétée autant de fois que l'utilisateur a de thème abonné, et rendant impossible cette fonctionnalité de pagination des résultats.

- Récupération de tous les topics

A l'inverse, il est peu probable que, même si un utilisateur s'abonne à une centaine de thème (ce qui est déjà improbable en soit), cela conduise à un vrai problème de performance au rendu de la liste complète des abonnements en même temps que les autres informations de l'utilisateur (l'objet associé étant très petit). De plus, il me semblait plus intuitif de gérer les abonnements et désabonnements sur une route « **topic/{id}/subscribe** » plutôt que « **user/topic/{topic id}** », et ainsi de réunir au mieux les routes et actions selon les ressources concernées.

Choix techniques côté front

Langage et framework

choix technique	lien vers le site / la documentation / une ressource	but du choix
Angular	https://angular.fr/ https://angular.dev/	Créer une application dynamique et évolutive.

Angular est un framework typescript offrant un développement front-end efficace, une très bonne complémentarité avec Spring Boot, et une application réactive aux interactions utilisateurs.

Librairies de test

choix technique	lien vers le site / la documentation / une ressource	but du choix
Jest	https://jestjs.io/fr/docs/getting-started	Facilite le développement de tests robustes.

Jest est simple à configurer avec Angular et permet d'écrire des tests clairs et maintenables. C'est une librairie légère, populaire, très documentée, évoluant rapidement et peu complexe à intégrer.

Côté front, l'utilisation de cette librairie fera donc gagner l'application en robustesse (assurant davantage sa disponibilité et son évolutivité), à moindre coût d'effort et de temps.

Autres dépendances

choix technique	lien vers le site / la documentation / une ressource	but du choix
Angular Material	https://material.angular.io/components/categories	Gagner en temps sur les designs.

Angular Material est une librairie qui fournit un grand nombre de composants visuels, avec une intégration simple et des designs modernes. Elle est régulièrement renouvelée, documentée, et à jour sur les normes en vigueur). Son utilisation permettra un gain de temps significatif tout en assurant une qualité visuelle et expérience utilisateur idéale.

Cependant, régulièrement le choix a été fait de créer un composant complètement spécifique à l'application, malgré l'existence d'un composant similaire dans la librairie.

Ce choix a été réalisé dans les cas suivants :

- Nécessité de personnalisation trop importante pour être adapté à la maquette, rendant plus pertinent de créer notre propre composant.
- Composant simple et rapide à réaliser par nous-même, nous assurant une meilleure maîtrise de notre composant.

Observables et économie de requêtes

choix technique	lien vers le site / la documentation / une ressource	but du choix
BehaviorSubject	https://angular.fr/services/behavior-subject	Stocker l'état des données côté client pour faire des économies de requêtes http.

L'utilisation de **BehaviorSubject** permet d'optimiser les performances de l'application en faisant l'économie de requêtes redondantes.

Un BehaviorSubject maintient l'état actuel des données côté client et permet de vérifier si l'information est déjà présente avant de déclencher une nouvelle requête. Conséquence de ce choix, toute mise à jour des informations côté base de données suppose de mettre à jour le BehaviorSubject concerné. Cette opération étant moins coûteuse que de relancer la requête il s'agit d'un gain de performance.

- Exemple sur les données utilisateur

Le SessionUserService stocke dans son « userSubject » les données de l'utilisateur : si un composant requiert ces informations, alors il interroge le userSubject et ne lance la requête GET « user » que s'il ne contient pas de données.

Lors de la mise à jour de l'email ou du username de l'utilisateur via la page de profil, userSubject est mis à jour et tous les composants utilisant ses informations (donc étant abonnés à userSubject) seront mis au courant de cette mise à jour.