

2025-04-01

Professeur : BELHOMME DIDIER

STAGE & ÉPREUVE INTÉGRÉE: STACK TECHNOLOGIQUE

NOM DU Projet : EYEWEB

VANHEES Vincent

Table des matières

[1. Langages et Environnement de Développement 3](#_Toc194439770)

[1.1 Backend 3](#_Toc194439771)

[1.1.1 Langage principal : Java 17 3](#_Toc194439772)

[1.1.2 Framework d’application : Spring Boot 3.x 3](#_Toc194439773)

[1.1.3 Gestion des dépendances : Maven 3](#_Toc194439774)

[1.1.4 Configuration externe : YAML et .env 3](#_Toc194439775)

[1.1.5 Environnement de persistance : JPA & Hibernate 4](#_Toc194439776)

[1.1.6 Lombok 4](#_Toc194439777)

[1.1.7 Sécurité JWT 4](#_Toc194439778)

[1.2 Frontend 4](#_Toc194439779)

[1.2.1 Langage principal : TypeScript 5](#_Toc194439780)

[1.2.2 Framework : Angular 16 5](#_Toc194439781)

[1.2.3 SCSS & Bootstrap 5](#_Toc194439782)

[1.2.4 RxJS 5](#_Toc194439783)

[1.2.5 Formulaires dynamiques 6](#_Toc194439784)

[1.2.6 Gestion des dépendances 6](#_Toc194439785)

[2. Frameworks, Librairies et Concepts Structurants 6](#_Toc194439786)

[2.1 Backend 6](#_Toc194439787)

[2.1.1 Spring Boot 6](#_Toc194439788)

[2.1.2 Spring Data JPA 7](#_Toc194439789)

[2.1.3 Spring Security + JWT 7](#_Toc194439790)

[2.1.4 Lombok 7](#_Toc194439791)

[2.1.5 MapStruct 7](#_Toc194439792)

[2.2 Frontend 8](#_Toc194439793)

[2.2.1 Angular 8](#_Toc194439794)

[2.2.2 PrimeNG 8](#_Toc194439795)

[2.2.3 RxJS 8](#_Toc194439796)

[2.2.4 Angular Forms 8](#_Toc194439797)

[2.2.5 Bootstrap & SCSS 9](#_Toc194439798)

[3. Base de Données et Persistance 9](#_Toc194439799)

[3.1 PostgreSQL 15 comme socle relationnel 9](#_Toc194439800)

[3.2 Mapping Objet-Relationnel (ORM) 9](#_Toc194439801)

[3.3 Audit, Historique, et Journalisation 9](#_Toc194439802)

[3.4 Stratégie de Chargement (EAGER vs LAZY) 10](#_Toc194439803)

[3.5 Scripts d’initialisation et migration 10](#_Toc194439804)

[3.6 Sécurité des données et accès 10](#_Toc194439805)

[3.7 Optimisations 10](#_Toc194439806)

[4. Sécurité & Authentification 10](#_Toc194439807)

[4.1 Vue d’ensemble du système de sécurité 10](#_Toc194439808)

[4.2 Composants clés de l’implémentation 11](#_Toc194439809)

[4.2.1 SecurityConfig.java 11](#_Toc194439810)

[4.2.2 JwtAuthenticationFilter.java 11](#_Toc194439811)

[4.3 Gestion des utilisateurs 11](#_Toc194439812)

[4.3.1 AuthenticationServiceImpl.java 11](#_Toc194439813)

[4.3.2 UserDetailsServiceImpl.java 11](#_Toc194439814)

[4.3.3 Token.java 12](#_Toc194439815)

[4.4 Génération et vérification des JWT 12](#_Toc194439816)

[4.4.1 JwtServiceImpl.java 12](#_Toc194439817)

[4.4.2 Méthodes : 12](#_Toc194439818)

[4.5 Contrôle d’accès par rôle 12](#_Toc194439819)

[4.6 Réinitialisation de mot de passe 12](#_Toc194439820)

[4.7 Cycle de vie d’une session utilisateur 13](#_Toc194439821)

[4.8 Sécurité renforcée 13](#_Toc194439822)

# Langages et Environnement de Développement

Ce chapitre présente les langages de programmation, les environnements de développement, les outils de configuration et les systèmes utilisés pour concevoir l’application EyeApp, tant du côté frontend que backend.

## Backend

Le backend de l’application EyeApp repose sur un environnement Java moderne, structuré autour du framework Spring Boot. Le développement est orienté API RESTful, respectant une architecture modulaire et sécurisée.

### Langage principal : Java 17

Le cœur du backend est développé en Java 17, une version LTS (Long Term Support) du langage. Elle apporte des fonctionnalités modernes comme les sealed classes et des performances accrues. Java a été choisi pour sa robustesse, sa compatibilité avec les frameworks Spring, et sa gestion stricte du typage, réduisant les erreurs d’exécution.

### Framework d’application : Spring Boot 3.x

Le backend repose sur Spring Boot, un framework Java qui facilite le développement d'applications web robustes en cachant une grande partie de la configuration XML classique. Il permet un lancement autonome de l’application via la classe EyeAppApplication.java et assure une architecture modulaire organisée autour des concepts Controller, Service, Repository, et Model.

La version utilisée, Spring Boot 3, prend en charge nativement Jakarta EE 10, avec les annotations jakarta.persistence.\*, et introduit des améliorations de performances ainsi que des optimisations liées à la sécurité.

### Gestion des dépendances : Maven

Les dépendances du projet backend sont gérées à l’aide du gestionnaire de build Apache Maven, via le fichier pom.xml. Ce fichier contient :

* Les dépendances essentielles (spring-boot-starter-web, spring-boot-starter-data-jpa, spring-boot-starter-security, etc.)
* Des outils de test (spring-boot-starter-test)
* Des bibliothèques de sécurité JWT (comme jjwt-api, jjwt-impl, jjwt-jackson)
* Des dépendances pour PostgreSQL (postgresql)
* Des plugins pour la compilation (maven-compiler-plugin avec Java 17)
* Des bibliothèques de mapping objet-DTO (MapStruct)
* Des bibliothèques de logging (spring-boot-starter-logging)

Maven assure une gestion centralisée, reproductible et automatisée des bibliothèques tierces.

### Configuration externe : YAML et .env

La configuration de l’application s’appuie sur :

* application.yaml pour la configuration générale du projet.
* application-dev.yaml pour les paramètres de développement (ex. : port, chemin de contexte, connexions à la base).
* .env pour les variables d’environnement (mot de passe admin, clef secrète JWT, etc.)

Ce découpage respecte les bonnes pratiques de séparation des environnements (dev, prod, test) et permet une configuration flexible, versionnée mais sécurisée.

### Environnement de persistance : JPA & Hibernate

Le backend utilise Spring Data JPA, basé sur Hibernate, pour effectuer l’ORM (Object-Relational Mapping) entre les entités Java (comme Patient, Exam, Calcul, etc.) et la base de données PostgreSQL.

Chaque entité est annotée avec :

* @Entity, @Table pour la persistance.
* @Id, @GeneratedValue pour l’identifiant primaire.
* @ManyToOne, @OneToMany pour les relations.
* @Column, @Enumerated et @JoinColumn pour les attributs complexes.

Hibernate permet la génération automatique des schémas (grâce à ddl-auto: update), tout en respectant la cohérence des types entre le Java et le SQL.

### Lombok

Le projet utilise Lombok, une bibliothèque Java qui génère automatiquement le code répétitif à la compilation :

* @Getter
* @Setter,
* @Builder,
* @NoArgsConstructor
* @AllArgsConstructor

sur les entités et DTOs.

Cela réduit la verbosité du code tout en améliorant la maintenabilité.

### Sécurité JWT

La couche de sécurité repose sur Spring Security et JWT, combinant :

* Des filtres (JwtAuthenticationFilter)
* Des configurations (SecurityConfig)
* Des services (AuthenticationServiceImpl, JwtServiceImpl, etc.)
* Une stratégie stateless avec stockage des tokens en base (TokenRepository)

## Frontend

Le frontend de EyeApp est développé avec Angular 16, un framework TypeScript moderne permettant de concevoir des applications web dynamiques de type SPA (Single Page Application). La structure est claire et orientée composants.

### Langage principal : TypeScript

L’application est écrite en TypeScript, un sur-ensemble de JavaScript typé, compilé avant exécution. TypeScript est intégré nativement à Angular et permet :

* Une meilleure robustesse du code
* L’introspection via l’IDE (auto-complétion, types)
* L’utilisation de classes, interfaces, décorateurs

Le fichier tsconfig.json configure les options de compilation, en spécifiant le niveau de stricteté, les chemins, les bibliothèques cibles, etc.

### Framework : Angular 16

Le cœur de l’interface est construit sur Angular 16, avec :

* Des modules fonctionnels (app.module.ts, app.config.module.ts)
* Des composants métiers (summary.component.ts, calculator.component.ts, etc.)
* Des routes déclaratives (app-routing.module.ts)
* Des services injectables (summary.service.ts, patient.service.ts, etc.)

Chaque page ou vue étant une composition de composants réutilisables, configurés par bindings et événements.

### SCSS & Bootstrap

Le style est défini en SCSS (via styles.scss), offrant une syntaxe plus avancée que CSS :

* Variables
* Imbrication de règle
* Mixins et héritages

Une partie du style repose sur Bootstrap (via classes SCSS), garantissant une mise en page responsive, fluide, et mobile-first.

1.2.4. UI Framework : PrimeNG

PrimeNG est utilisé pour les composants UI :

* p-table : tableaux dynamiques
* p-dialog : boîtes de dialogue
* p-tree : arbres de sélection hiérarchique
* p-toast : notifications
* p-dropdown, p-calendar, etc.

Les composants sont intégrés dans les fichiers .component.ts et .component.html, en association avec Angular Forms et le modèle RxJS.

### RxJS

Le projet utilise RxJS pour la gestion réactive des données :

* Observable<T> pour représenter les flux asynchrones.
* subscribe()
* pipe()
* map()
* tap()

pour traiter les flux. Cela permet de gérer les appels HTTP, les changements d’état, les formulaires, et les traitements dynamiques.

### Formulaires dynamiques

Le frontend utilise des Reactive Forms (via FormBuilder, FormGroup, Validators) pour :

* Créer dynamiquement les champs
* Valider les entrées en temps réel
* Réagir aux événements utilisateurs

Cela garantit un contrôle typé des formulaires, avec des erreurs personnalisables.

### Gestion des dépendances

Le frontend utilise npm via le fichier package.json :

* Déclaration des packages : @angular/core, primeng, rxjs, bootstrap, etc.
* Scripts de build : ng serve, ng build, etc.
* Fichier package-lock.json pour le verrouillage des versions

Le système garantit une construction déterministe du projet, sur tout poste de développement.

# Frameworks, Librairies et Concepts Structurants

## Backend

### Spring Boot

Spring Boot simplifie la création d’applications Java en encapsulant une configuration conventionnelle et en réduisant le code boilerplate. Dans ce projet, Spring Boot est utilisé pour :

* Créer des contrôleurs REST (@RestController) pour exposer les endpoints de l’API.
* Injecter les dépendances avec @Autowired ou constructeur.
* Définir les beans de configuration avec @Configuration, @Bean.
* Fournir un serveur Tomcat embarqué, sans configuration manuelle d’un conteneur de servlets externe.
* Gérer le cycle de vie de l'application via EyeAppApplication.java.

Les annotations comme @SpringBootApplication fusionnent plusieurs fonctionnalités (@Configuration, @EnableAutoConfiguration, @ComponentScan).

### Spring Data JPA

Le projet exploite Spring Data JPA pour gérer la persistance des entités via des interfaces Repository :

* JpaRepository<Entité, Long> : donne accès aux opérations CRUD, findAll, save, deleteById, etc.
* Méthodes dérivées : ex. findByLastname(String lastname) sont automatiquement traduites en requêtes SQL.
* Requêtes personnalisées avec @Query, ex. findSelectedExamByPatientId(Long patientId).

Les entités Java (Exam, Patient, Calcul, Lens, etc.) sont annotées avec @Entity, et mappées vers les tables PostgreSQL. Les relations sont exprimées par @ManyToOne, @OneToMany, @JoinColumn, garantissant l’intégrité des données.

Hibernate est le moteur ORM sous-jacent à Spring Data JPA, permettant la génération de SQL au runtime.

### Spring Security + JWT

La sécurité du backend repose sur l’alliance de Spring Security 6 et JWT (JSON Web Tokens). Cela permet :

* Une gestion fine des accès aux endpoints API.
* Une authentification stateless.
* Une séparation stricte entre les utilisateurs (Admins, Médecins, Secrétaires...).

Composants clés :

* SecurityConfig : définition des routes publiques, privées, et des filtres.
* JwtAuthenticationFilter : extraction et validation du JWT.
* AuthenticationServiceImpl, JwtServiceImpl, LogoutServiceImpl : services métiers de sécurité.
* UserDetailsServiceImpl : intégration avec Spring Security pour charger l’utilisateur courant.
* TokenRepository : stockage des tokens en base (expirés, révoqués, valides).

L’intégralité du cycle d’authentification/autorisation repose sur JWT signé, injecté dans le header HTTP (Authorization: Bearer <token>).

### Lombok

Lombok est utilisé pour alléger le code des entités et DTOs. Il permet :

* Génération automatique de getters/setters (@Getter, @Setter).
* Constructeurs (@NoArgsConstructor, @AllArgsConstructor).
* @Builder : design pattern de création d’objet.
* @SuperBuilder : utilisé pour hériter d’un modèle commun (IdentifiedModel).

Cela contribue à une meilleure lisibilité du code et à une productivité accrue.

### MapStruct

Bien que peu visible dans les fichiers actuels, le projet est prêt pour intégrer MapStruct afin de mapper les entités vers les DTOs (SummaryDto, etc.).

@Mapper : interface de mappage (ex. SummaryMapper).

@Mapping : permet de spécifier les correspondances entre champs.

Cette bibliothèque compile du code de mapping performant, sans coût au runtime.

## Frontend

### Angular

Angular 16 est le framework frontend du projet. Il repose sur des principes de :

* Architecture modulaire : découpage par feature/module (AppModule, AppConfigModule...).
* Composants réutilisables : chaque vue est représentée par un composant Angular (Component avec ses fichiers .ts, .html, .scss).
* Injection de dépendances : Angular injecte les services (PatientService, SummaryService, etc.) de façon transparente.
* Routing : la navigation SPA est gérée avec RouterModule, en configurant les routes vers les composants (home, summary, calculator, etc.).
* Les métadonnées décorées par @Component, @Injectable, @NgModule facilitent une architecture solide, maintenable et évolutive.

### PrimeNG

PrimeNG fournit les composants UI utilisés dans EyeApp. Ce framework offre :

* Richesse visuelle : tableaux dynamiques (p-table), arbres (p-tree), dialogues (p-dialog), filtres (p-columnFilter), etc.
* Compatibilité Angular : intégration parfaite avec Angular Forms.
* Accessibilité et responsivité : composants adaptatifs avec peu de configuration CSS additionnelle.
* La personnalisation graphique s’appuie sur theme.scss, hérité de PrimeNG, et intégré dans styles.scss.

### RxJS

Le projet utilise RxJS (Reactive Extensions for JavaScript) pour manipuler les flux de données :

* Observable<T> : représentation d’un flux de données asynchrone.
* Opérateurs pipe(), map(), tap(), catchError() : permettent de transformer, sécuriser et enchaîner les traitements.

RxJS a une présence constante dans la gestion des appels HTTP (HttpClient.get/post/put) et dans les formulaires réactifs.

### Angular Forms

L’interface utilise les formulaires réactifs :

* FormGroup, FormControl, Validators
* formGroup, formControlName dans les templates
* Validation dynamique des champs, affichage des messages d’erreurs

Ces formulaires sont typés, testables, et très bien intégrés avec PrimeNG.

### Bootstrap & SCSS

L’application est stylisée avec un mélange de :

* SCSS : centralisation des variables, modularité, imbrication (styles.scss)
* Bootstrap : classes utilitaires pour la mise en page (grids, buttons, spacing)
* Le design est responsive et épuré, adapté à un usage desktop/tablette.

# Base de Données et Persistance

## PostgreSQL 15 comme socle relationnel

Le choix de PostgreSQL répond à plusieurs exigences du domaine médical :

* Fiabilité transactionnelle : chaque opération est atomique, cohérente, isolée et durable.
* Puissance relationnelle : permet de modéliser des entités complexes avec relations 1-N, N-N.
* Indexation avancée : utilisé pour améliorer les performances sur des champs comme id, patient\_id, exam\_id.
* Extensions médicales possibles : support potentiel pour des types comme hstore, jsonb, tsvector pour le futur.
* @JoinColumn(name = "...") : gère manuellement le nom des clés étrangères. En base PostgreSQL, cela se traduit par des clés étrangères classiques, avec contraintes ON DELETE CASCADE ou SET NULL en fonction des besoins métier.

## Mapping Objet-Relationnel (ORM)

Le projet utilise Spring Data JPA combiné à Hibernate pour faire le pont entre :

* Entités Java (Exam.java, Patient.java, Calcul.java, Lens.java, etc.)
* Tables relationnelles dans PostgreSQL

Chaque entité :

* Est annotée avec @Entity, @Table(name = "...")
* Hérite d’un modèle IdentifiedModel contenant l’id, createdAt, modifiedAt, deleted, facilitant la journalisation et le versioning.
* Dispose de getters/setters générés via Lombok (@Getter, @Setter), et de constructeurs adaptés (@NoArgsConstructor, @AllArgsConstructor).

## Audit, Historique, et Journalisation

Le projet active le versioning automatique des entités avec :

* @Audited (Hibernate Envers) : permet de conserver un historique des modifications de chaque entité.
* Table audit automatiquement générée par Hibernate pour chaque entité auditée.

Cela est essentiel dans un contexte médical pour :

* Tracer les modifications de données sensibles.
* Justifier les interventions.
* Respecter les normes RGPD et médicales.

## Stratégie de Chargement (EAGER vs LAZY)

Par défaut, la plupart des relations sont configurées avec :

* @ManyToOne(fetch = FetchType.LAZY) : pour éviter de charger les objets liés inutilement.
* @OneToMany(fetch = FetchType.LAZY) : les collections (calculs, liens) sont chargées à la demande.

Cela réduit la consommation mémoire et limite les requêtes complexes sauf si explicitement appelées.

## Scripts d’initialisation et migration

La base est initialisée via des entités Java directement persistées par SpringBootApplication. Aucune trace d’outils de migration type Flyway ou Liquibase n’est détectée, mais l’extension est possible :

* spring.jpa.hibernate.ddl-auto=update : permet de créer automatiquement les tables lors du premier lancement.
* Adapté au développement, mais à remplacer en production par une solution de migration versionnée.

## Sécurité des données et accès

La sécurité en base repose sur deux niveaux :

Côté applicatif

* Filtres Spring Security empêchant l’accès aux endpoints non autorisés
* @PreAuthorize, @Secured sur les services et contrôleurs

Côté base

* Accès restreint via credentials stockés dans application.yaml
* Utilisation d’un utilisateur postgres ou eyeapp\_user avec droits limités
* Données sensibles comme Token, VerificationCode sont encryptées et protégées

## Optimisations

Le modèle respecte plusieurs principes :

* 3e forme normale (3NF) : aucune redondance, relations bien définies
* Index implicites : sur toutes les clés primaires et étrangères
* Tri et pagination côté JPA : via Pageable dans les services si besoin
* Partitionnement logique : à travers les entités (Exam, Calcul, Lens, etc.)

# Sécurité & Authentification

La sécurité est basée sur Spring Security 6, JWT (JSON Web Tokens), et une gestion personnalisée des utilisateurs, des rôles et des sessions. L’idée est de garantir que seules les personnes authentifiées et autorisées puissent accéder aux fonctionnalités et données sensibles de l’application médicale.

## Vue d’ensemble du système de sécurité

Objectifs principaux :

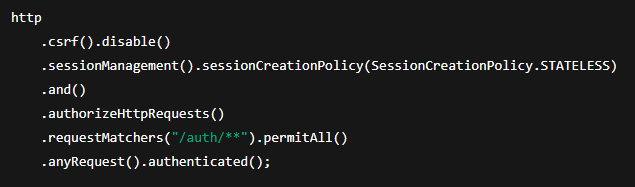
* Authentification forte avec JWT (stateless, sécurisée, portable)
* Séparation nette entre utilisateurs authentifiés et anonymes
* Vérification systématique de chaque requête HTTP entrante
* Gestion dynamique des rôles (ROLE\_ADMIN, ROLE\_MANAGER, ROLE\_CLIENT)
* Révocation possible de tokens et gestion multi-sessions
* Système de vérification par code de sécurité (reset password, validation)

## Composants clés de l’implémentation

### SecurityConfig.java

Cette classe configure toute la stratégie de sécurité de l’application :

* Active CORS, désactive CSRF
* Configure la politique sessionCreationPolicy = STATELESS
* Autorise les endpoints publics : /auth/\*\*
* Protège tous les autres endpoints (authenticated())
* Enregistre le filtre JwtAuthenticationFilter avant UsernamePasswordAuthenticationFilter
* Gère les rôles avec GrantedAuthority



### JwtAuthenticationFilter.java

* Intercepte chaque requête HTTP entrante
* Recherche un JWT dans l’en-tête Authorization: Bearer <token>
* Valide le token via JwtServiceImpl
* Si valide : charge l’utilisateur via UserDetailsServiceImpl et l’insère dans le SecurityContextHolder
* Sinon : laisse passer la requête sans authentification (accès refusé aux endpoints protégés)

## Gestion des utilisateurs

### AuthenticationServiceImpl.java

Fournit les méthodes :

* register() : crée un utilisateur, encode son mot de passe, génère un JWT, enregistre le token
* authenticate() : vérifie les credentials, retourne un nouveau token
* logout() : révoque les tokens actifs (via TokenRepository)

### UserDetailsServiceImpl.java

* Charge un utilisateur (User) depuis la base de données PostgreSQL
* Renvoie un UserDetails enrichi avec ses rôles Spring (GrantedAuthority)
* C’est ce composant qui fournit les informations à Spring Security pour appliquer les règles d’autorisation

### Token.java

Entité persistante représentant chaque JWT émis

Contient : token, revoked, expired, user

Permet :

* De vérifier dynamiquement si un token est toujours valide
* De révoquer un token manuellement (logout)
* De gérer plusieurs tokens par utilisateur (sessions multiples)

## Génération et vérification des JWT

### JwtServiceImpl.java

Service central qui manipule les JWT :

Clé secrète (secret-key) définie dans application.yaml

### Méthodes :

* generateToken(UserDetails user) : création d’un token signé avec claims
* isTokenValid(token, user) : vérifie la validité, la signature, la date d’expiration
* extractUsername(token) : extrait l’e-mail pour identifier l’utilisateur
* extractClaim() : accède à n’importe quelle valeur embarquée (ex : rôle, iat, exp)

## Contrôle d’accès par rôle

Rôles définis dans l’entité User.java via @Enumerated(EnumType.STRING)

Les contrôleurs REST utilisent :

* @PreAuthorize("hasRole('ROLE\_ADMIN')")
* Ou bien la protection globale par filtre
* La résolution des rôles est automatique lors du parsing du JWT (CustomAuthorityDeserializer)

## Réinitialisation de mot de passe

Composants :

* ResetPasswordRequest.java : contient l’e-mail et le nouveau mot de passe
* VerificationCode.java : entité représentant un code de validation (ex: 6 chiffres)
* L’utilisateur reçoit un code (par e-mail), et doit le fournir pour finaliser la réinitialisation
* Le code est temporaire et lié à un compte spécifique

Toute tentative de validation échoue si :

* Code expiré
* Mauvais code
* Utilisateur inexistant

## Cycle de vie d’une session utilisateur

Inscription (POST /auth/register)

* L’utilisateur envoie un JSON avec email, nom, mot de passe, rôle
* Mot de passe hashé (PasswordEncoder)
* Enregistrement dans UserRepository
* Génération et stockage du JWT (stateless)
* Connexion (POST /auth/authenticate)

Vérification du mot de passe

* Si valide, enregistrement d’un nouveau JWT en base
* Appels sécurisés
* Frontend ajoute Authorization: Bearer <token>
* Filtre JwtAuthenticationFilter valide le token
* Spring Security autorise ou non la requête

Déconnexion

* Appel à /auth/logout
* Révocation de tous les tokens valides en base (token.setRevoked(true))

## Sécurité renforcée

* JWT stockés en base = contrôle dynamique possible
* Pas de session HTTP = architecture totalement stateless
* @Transactional dans les services = cohérence transactionnelle
* Logger intégré dans les services = journalisation des connexions