

Filière : Bachelier en informatique de gestion

Projet de fin d’étude



Encadré par :

**Didier Belhomme**

Présenté par :

**Vincent Vanhees**

Faculté des Lorem Ipsum est simplement du faux texte employé

Année scolaire 2024 – 2025

Table des matières

[1. Introduction 8](#_Toc194609405)

[2. Langages et Environnement de Développement 8](#_Toc194609406)

[2.1 Backend 9](#_Toc194609407)

[2.1.1 Langage principal : Java 17 9](#_Toc194609408)

[2.1.2 Framework d’application : Spring Boot 3.x 9](#_Toc194609409)

[2.1.3 Gestion des dépendances : Maven 9](#_Toc194609410)

[2.1.4 Configuration externe : YAML et .env 9](#_Toc194609411)

[2.1.5 Environnement de persistance : JPA & Hibernate 10](#_Toc194609412)

[2.1.6 Lombok 10](#_Toc194609413)

[2.1.7 Sécurité JWT 10](#_Toc194609414)

[2.2 Frontend 10](#_Toc194609415)

[2.2.1 Langage principal : TypeScript 10](#_Toc194609416)

[2.2.2 Framework : Angular 16 11](#_Toc194609417)

[2.2.3 SCSS & Bootstrap 11](#_Toc194609418)

[2.2.4 RxJS 11](#_Toc194609419)

[2.2.5 Formulaires dynamiques 12](#_Toc194609420)

[2.2.6 Gestion des dépendances 12](#_Toc194609421)

[3. Frameworks, Librairies et Concepts Structurants 12](#_Toc194609422)

[3.1 Backend 12](#_Toc194609423)

[3.1.1 Spring Boot 12](#_Toc194609424)

[3.1.2 Spring Data JPA 12](#_Toc194609425)

[3.1.3 Spring Security + JWT 13](#_Toc194609426)

[3.1.4 Lombok 13](#_Toc194609427)

[3.1.5 MapStruct 13](#_Toc194609428)

[3.2 Frontend 14](#_Toc194609429)

[3.2.1 Angular 14](#_Toc194609430)

[3.2.2 PrimeNG 14](#_Toc194609431)

[3.2.3 RxJS 14](#_Toc194609432)

[3.2.4 Angular Forms 14](#_Toc194609433)

[3.2.5 Bootstrap & SCSS 14](#_Toc194609434)

[4. Base de Données et Persistance 15](#_Toc194609435)

[4.1 PostgreSQL 15 comme socle relationnel 15](#_Toc194609436)

[4.2 Mapping Objet-Relationnel (ORM) 15](#_Toc194609437)

[4.3 Audit, Historique, et Journalisation 15](#_Toc194609438)

[4.4 Stratégie de Chargement (EAGER vs LAZY) 15](#_Toc194609439)

[4.5 Scripts d’initialisation et migration 16](#_Toc194609440)

[4.6 Sécurité des données et accès 16](#_Toc194609441)

[4.7 Optimisations 16](#_Toc194609442)

[5. Sécurité & Authentification 16](#_Toc194609443)

[5.1 Vue d’ensemble du système de sécurité 16](#_Toc194609444)

[5.2 Composants clés de l’implémentation 17](#_Toc194609445)

[5.2.1 SecurityConfig.java 17](#_Toc194609446)

[5.2.2 JwtAuthenticationFilter.java 17](#_Toc194609447)

[5.3 Gestion des utilisateurs 17](#_Toc194609448)

[5.3.1 AuthenticationServiceImpl.java 17](#_Toc194609449)

[5.3.2 UserDetailsServiceImpl.java 17](#_Toc194609450)

[5.3.3 Token.java 17](#_Toc194609451)

[5.4 Génération et vérification des JWT 18](#_Toc194609452)

[5.4.1 JwtServiceImpl.java 18](#_Toc194609453)

[5.5 Contrôle d’accès par rôle 18](#_Toc194609454)

[5.6 Réinitialisation de mot de passe 18](#_Toc194609455)

[5.7 Cycle de vie d’une session utilisateur 18](#_Toc194609456)

[5.8 Sécurité renforcée 19](#_Toc194609457)

[6. Architecture générale du projet EyeWeb 19](#_Toc194609458)

[6.1 Organisation modulaire des couches 19](#_Toc194609459)

[6.1.1 Controller layer 19](#_Toc194609460)

[6.1.2 Service layer 19](#_Toc194609461)

[6.1.3 Repository layer 20](#_Toc194609462)

[6.1.4 Entity layer 20](#_Toc194609463)

[6.1.5 DTOs & Mappers 20](#_Toc194609464)

[6.2 Technologies utilisées par couche 20](#_Toc194609465)

[6.3 Particularités de l’architecture 20](#_Toc194609466)

[6.3.1 Approche API First 20](#_Toc194609467)

[6.3.2 Architecture RESTful 20](#_Toc194609468)

[6.4 Architecture du frontend Angular 21](#_Toc194609469)

[6.4.1 Structure des fichiers Angular 21](#_Toc194609470)

[6.4.2 Services Angular 21](#_Toc194609471)

[6.4.3 Routage Angular 21](#_Toc194609472)

[6.4.4 Forms & Reactive Forms 21](#_Toc194609473)

[6.5 Services transversaux 22](#_Toc194609474)

[7. Frontend : Composants, flux de données, services et expérience utilisateur 22](#_Toc194609475)

[7.1 Structure générale du frontend Angular 22](#_Toc194609476)

[7.2 Composants personnalisés 23](#_Toc194609477)

[7.3 Système de routage 23](#_Toc194609478)

[7.4 Services Angular & Flux RxJS 23](#_Toc194609479)

[7.5 Gestion des formulaires 24](#_Toc194609480)

[7.6 Bibliothèques UI : PrimeNG 24](#_Toc194609481)

[7.7 UI, UX et accessibilité 25](#_Toc194609482)

[7.8 Sécurité dans Angular 25](#_Toc194609483)

[7.9 Compilation, packaging et configuration 25](#_Toc194609484)

[8. Base de données PostgreSQL et ORM (JPA / Hibernate) 25](#_Toc194609485)

[8.1 Structure générale 25](#_Toc194609486)

[8.2 Mapping objet-relationnel (ORM) 26](#_Toc194609487)

[8.3 Requêtes personnalisées (JPQL) 26](#_Toc194609488)

[8.4 Optimisation : 26](#_Toc194609489)

[9. Communication API REST et respect de l’approche API-First 26](#_Toc194609490)

[9.1 Standards REST 26](#_Toc194609491)

[9.2 Contrôleurs REST (@RestController) 26](#_Toc194609492)

[9.3 DTOs et Mappers 27](#_Toc194609493)

[9.4 Services 27](#_Toc194609494)

[10. Tests, qualité du code et outils 27](#_Toc194609495)

[10.1 Tests unitaires et d’intégration (Backend) 27](#_Toc194609496)

[10.2 Tests frontend 27](#_Toc194609497)

[10.3 Qualité de code 27](#_Toc194609498)

[10.4 CI/CD 27](#_Toc194609499)

[11. Gestion des rôles et des droits (RBAC - Role-Based Access Control) 27](#_Toc194609500)

[11.1 Définition des rôles 28](#_Toc194609501)

[11.2 Stockage et affectation 28](#_Toc194609502)

[11.3 Chargement via UserDetailsService 28](#_Toc194609503)

[11.4 Application des droits dans les endpoints 28](#_Toc194609504)

[11.5 Vérification côté service 29](#_Toc194609505)

[11.6 Avantages de RBAC dans l'application 29](#_Toc194609506)

[12. Gestion des erreurs, des performances et des logs 29](#_Toc194609507)

[12.1 Gestion des erreurs dans le backend 29](#_Toc194609508)

[12.2 Formats de réponse d'erreurs (DTO) 30](#_Toc194609509)

[12.3 Journalisation avec SLF4J / Logback 30](#_Toc194609510)

[12.4 Logs de sécurité et d’audit 30](#_Toc194609511)

[12.5 Gestion des performances 30](#_Toc194609512)

[12.6 Analyse future 30](#_Toc194609513)

[13. Système de routing Angular et navigation entre composants 31](#_Toc194609514)

[13.1 Initialisation du routing 31](#_Toc194609515)

[13.2 Composants liés au routing 31](#_Toc194609516)

[13.3 Navigation entre pages 31](#_Toc194609517)

[13.4 Protection des routes : AuthGuard 32](#_Toc194609518)

[13.5 Paramètres de route (ActivatedRoute) 32](#_Toc194609519)

[13.6 Routes enfant et modules 32](#_Toc194609520)

[13.7 Navigation par lien (template HTML) 32](#_Toc194609521)

[13.8 Gestion des erreurs et redirections 33](#_Toc194609522)

[14. Services Angular et gestion des données via RxJS et HTTPClient 33](#_Toc194609523)

[14.1 Rôle des services dans l'architecture Angular 33](#_Toc194609524)

[14.2 Appels API : HTTPClientModule 33](#_Toc194609525)

[14.3 Gestion des Observables et RxJS 34](#_Toc194609526)

[14.4 Injection de services et dépendances 34](#_Toc194609527)

[14.5 Services métiers disponibles dans EyeWeb 34](#_Toc194609528)

[14.6 Traitement des erreurs globales 35](#_Toc194609529)

[14.7 Intégration des services dans les composants 35](#_Toc194609530)

[14.8 Services transversaux 35](#_Toc194609531)

[14.9 Avantages de cette architecture de services 36](#_Toc194609532)

[15. Structure des Composants Angular : Organisation, Interaction et Bonnes Pratiques 36](#_Toc194609533)

[15.1 Anatomie d’un composant Angular 36](#_Toc194609534)

[15.2 Hiérarchie et découpage des composants 37](#_Toc194609535)

[15.3 Cycle de vie Angular utilisé 37](#_Toc194609536)

[15.4 Communication entre composants 37](#_Toc194609537)

[15.5 Modularité des composants 38](#_Toc194609538)

[15.6 Chargement dynamique de composants 38](#_Toc194609539)

[15.7 Interaction avec l’interface (UI) 38](#_Toc194609540)

[15.8 Accessibilité et Responsiveness 38](#_Toc194609541)

[15.9 Tests des composants (structure) 38](#_Toc194609542)

[15.10 1Cas d’usage concrets dans EyeWeb 39](#_Toc194609543)

[16. Routing et navigation dans EyeWeb 39](#_Toc194609544)

[16.1 Objectifs du système de routing 39](#_Toc194609545)

[16.2 Structure du routing principal 39](#_Toc194609546)

[16.3 Lazy loading des modules 40](#_Toc194609547)

[16.4 Sécurisation des routes avec AuthGuard 40](#_Toc194609548)

[16.5 Passage de paramètres dans l’URL 41](#_Toc194609549)

[16.6 Navigation dans le code 41](#_Toc194609550)

[16.7 Liens dans les templates (HTML) 42](#_Toc194609551)

[16.8 Redirections et pages 404 42](#_Toc194609552)

[16.9 Synchronisation avec le backend 43](#_Toc194609553)

[17. Services Angular et consommation d’API REST 43](#_Toc194609554)

[17.1 Rôle des services Angular 43](#_Toc194609555)

[17.2 Structure des services dans le projet 43](#_Toc194609556)

[17.3 Exemple de service standardisé 43](#_Toc194609557)

[17.4 Sécurité des requêtes HTTP 44](#_Toc194609558)

[17.5 Utilisation dans les composants 45](#_Toc194609559)

[17.6 Gestion centralisée des erreurs 45](#_Toc194609560)

[17.7 Appels conditionnels et typage avancé 45](#_Toc194609561)

[17.8 Appels spécifiques : calcul, résumé, auth 46](#_Toc194609562)

[17.9 Séparation des couches et bonnes pratiques 46](#_Toc194609563)

[18. Composants, interfaces et gestion des états 46](#_Toc194609564)

[18.1 Organisation des composants Angular 46](#_Toc194609565)

[18.2 Structure d’un composant Angular 47](#_Toc194609566)

[18.3 Interfaces TypeScript fortement typées 48](#_Toc194609567)

[18.4 Liaison de données et formulaire réactif 48](#_Toc194609568)

[18.5 Rendu conditionnel et composants dynamiques 49](#_Toc194609569)

[18.6 Communication entre composants 49](#_Toc194609570)

[18.7 État local et global 50](#_Toc194609571)

[18.8 Gestion des erreurs dans les composants 50](#_Toc194609572)

[18.9 Intégration visuelle et design 50](#_Toc194609573)

[18.10 Résumé 51](#_Toc194609574)

[19. Routing, Navigation et Modules Angular 51](#_Toc194609575)

[19.1 Configuration du Routing principal (app-routing.module.ts) 51](#_Toc194609576)

[19.2 Modules Angular et découpage fonctionnel 52](#_Toc194609577)

[19.3 Lazy Loading et séparation des responsabilités 52](#_Toc194609578)

[19.4 Gestion de la navigation (Router, ActivatedRoute) 52](#_Toc194609579)

[19.5 Routage conditionnel et redirections 53](#_Toc194609580)

[19.6 AuthGuard et accès sécurisé 53](#_Toc194609581)

[19.7 Routing dans les modules internes 54](#_Toc194609582)

[19.8 Animation de transitions (optionnel) 54](#_Toc194609583)

[19.9 Navigation dans le HTML 54](#_Toc194609584)

[19.10 Résumé 55](#_Toc194609585)

[20. Services, Communication HTTP & Gestion des Erreurs dans Angular 55](#_Toc194609586)

[20.1 Architecture Générale des Services Angular 55](#_Toc194609587)

[20.2 Communication HTTP : le rôle du HttpClient 56](#_Toc194609588)

[20.3 Gestion des Observables avec RxJS 56](#_Toc194609589)

[20.4 Gestion Centralisée des Erreurs 57](#_Toc194609590)

[20.5 Sécurisation des requêtes HTTP avec JWT 57](#_Toc194609591)

[20.6 Services spécialisés et logique métier 58](#_Toc194609592)

[20.7 Détection d’erreurs frontend vs backend 58](#_Toc194609593)

[20.8 Utilisation dans les composants 58](#_Toc194609594)

[20.9 Évolution possible vers une couche de "store" 59](#_Toc194609595)

[20.10 Résumé 59](#_Toc194609596)

[21. Gestion des Formulaires Réactifs et Validation dans Angular 59](#_Toc194609597)

[21.1 Principe des Formulaires Réactifs 60](#_Toc194609598)

[21.2 Mise en place dans EyeWeb 60](#_Toc194609599)

[21.3 Création et structure d’un formulaire 60](#_Toc194609600)

[21.4 Validation des champs 61](#_Toc194609601)

[21.4.1 Validateurs Intégrés Angular 61](#_Toc194609602)

[21.4.2 Validateurs personnalisés 61](#_Toc194609603)

[21.5 Affichage des messages d’erreur 62](#_Toc194609604)

[21.6 Soumission du formulaire 62](#_Toc194609605)

[21.7 Formulaires dynamiques et FormArray 62](#_Toc194609606)

[21.8 Liaison bidirectionnelle avec les modèles 63](#_Toc194609607)

[21.9 Interactions conditionnelles dans le formulaire 63](#_Toc194609608)

[21.10 Résumé 63](#_Toc194609609)

[22. Routing, Navigation et Sécurisation des Routes dans Angular 64](#_Toc194609610)

[22.1 Structure de base du routing 64](#_Toc194609611)

[22.2 Navigation dynamique dans le code 64](#_Toc194609612)

[22.3 Paramètres de route 65](#_Toc194609613)

[22.4 Guards et sécurisation du routing 65](#_Toc194609614)

[22.5 Redirections et routes par défaut 66](#_Toc194609615)

[22.6 Lazy loading et modules enfants 66](#_Toc194609616)

[22.7 Synchronisation du routing et de l’état 66](#_Toc194609617)

[22.8 Navigation conditionnelle dans l’interface 66](#_Toc194609618)

[22.9 Avantages du système de routing Angular dans EyeWeb 67](#_Toc194609619)

[22.10 Bonnes pratiques adoptées 67](#_Toc194609620)

[23. Architecture des Services Angular : Communication avec l’API REST 67](#_Toc194609621)

[23.1 Structure générale des services 67](#_Toc194609622)

[23.2 Injection de HttpClient 68](#_Toc194609623)

[23.3 Exemple typique de service : Summary 68](#_Toc194609624)

[23.4 Typage strict et usage des modèles 68](#_Toc194609625)

[23.5 Observables et Réactivité 69](#_Toc194609626)

[23.6 Sécurité des requêtes : JWT dans HttpInterceptor 69](#_Toc194609627)

[23.7 Résilience : gestion des erreurs HTTP 70](#_Toc194609628)

[23.8 Cas particuliers de traitement : login et registration 70](#_Toc194609629)

[23.9 Architecture REST First : respect des verbes HTTP 70](#_Toc194609630)

[23.10 Environnement, modularité et centralisation 71](#_Toc194609631)

[23.11 Résumé des avantages du système de services dans EyeWeb 71](#_Toc194609632)

[24. Architecture et fonctionnement des composants Angular 71](#_Toc194609633)

[24.1 Introduction au système de composants Angular 71](#_Toc194609634)

[24.2 Organisation modulaire des composants 72](#_Toc194609635)

[24.3 Schéma MVC côté client 72](#_Toc194609636)

[24.4 Composants réactifs avec RxJS 72](#_Toc194609637)

[24.5 iaison de données (Data Binding) 73](#_Toc194609638)

[24.6 Passage de données entre composants 73](#_Toc194609639)

[24.7 Affichage conditionnel et boucles 73](#_Toc194609640)

[24.8 Formulaires dynamiques et validation 74](#_Toc194609641)

[24.9 Design UI avec PrimeNG 74](#_Toc194609642)

[24.10 Routage Angular entre composants 74](#_Toc194609643)

[24.11 Modularité & réutilisabilité 75](#_Toc194609644)

[24.12 Détection des changements 75](#_Toc194609645)

[24.13 Résumé des bonnes pratiques appliquées 75](#_Toc194609646)

[25. Flux de données et gestion des états dans l’application Angular 76](#_Toc194609647)

[25.1 Objectif du flux de données 76](#_Toc194609648)

[25.2 Principes de gestion d’état Angular 76](#_Toc194609649)

[25.3 Exemple de flux complet dans EyeWeb 76](#_Toc194609650)

[25.4 Utilisation des Observable et Subject 77](#_Toc194609651)

[25.5 Synchronisation composant-service 77](#_Toc194609652)

[25.6 Système de notifications (flux à part) 78](#_Toc194609653)

[25.7 État local vs global 78](#_Toc194609654)

[25.8 Usage avancé : switchMap, combineLatest, map 78](#_Toc194609655)

[25.9 Recommandations appliquées dans EyeWeb 79](#_Toc194609656)

[25.10 Conclusion du flux RxJS dans EyeWeb 79](#_Toc194609657)

[26. Système de Formulaires Angular dans EyeWeb 79](#_Toc194609658)

[26.1 Rôle central des formulaires 79](#_Toc194609659)

[26.2 Déclaration du FormGroup 79](#_Toc194609660)

[26.3 Affichage dans le template HTML 80](#_Toc194609661)

[26.4 Contrôles et validation 80](#_Toc194609662)

[26.5 Soumission du formulaire 81](#_Toc194609663)

[26.6 Mise à jour (patching) d’un formulaire 81](#_Toc194609664)

[26.7 Dynamisme dans les champs 81](#_Toc194609665)

[26.8 Intégration avec PrimeNG 82](#_Toc194609666)

[26.9 Messages de validation et feedback utilisateur 82](#_Toc194609667)

[26.10 Bonnes pratiques respectées dans EyeWeb 82](#_Toc194609668)

[27. Module Summary (Vue opératoire du patient) 83](#_Toc194609669)

[27.1 Objectif et rôle du module 83](#_Toc194609670)

[27.2 Architecture du module 83](#_Toc194609671)

[27.3 Service summary.service.ts 83](#_Toc194609672)

[27.4 Composant summary.component.ts 84](#_Toc194609673)

[27.5 Fichier HTML (summary.component.html) 84](#_Toc194609674)

[27.6 Intégration avec le routing 85](#_Toc194609675)

[27.7 Couplage avec PrimeNG 85](#_Toc194609676)

[27.8 Avantages de cette approche 85](#_Toc194609677)

[27.9 Lien avec le backend 86](#_Toc194609678)

[27.10 Perspectives d'évolution 86](#_Toc194609679)

[28. Module Patient (Gestion des patients) 86](#_Toc194609680)

[28.1 Objectif fonctionnel 86](#_Toc194609681)

[28.2 Structure modulaire (Angular) 86](#_Toc194609682)

[28.3 Fonctionnalités couvertes 87](#_Toc194609683)

[28.4 Intégration des composants PrimeNG 87](#_Toc194609684)

[28.5 Formulaire de patient (patient-form.component) 88](#_Toc194609685)

[28.6 Service Angular PatientService 88](#_Toc194609686)

[28.7 Routage Angular 89](#_Toc194609687)

[28.8 Connexion avec les autres modules 90](#_Toc194609688)

[28.9 Évolutions futures possibles 90](#_Toc194609689)

[29. Module Calculator (Calculs biométriques et sélection de l’IOL) 90](#_Toc194609690)

[29.1 Objectif fonctionnel 90](#_Toc194609691)

[29.2 Intégration dans le workflow patient 90](#_Toc194609692)

[29.3 Structure modulaire Angular 91](#_Toc194609693)

[29.4 Fonctionnalités métier 91](#_Toc194609694)

[29.5 Intégration backend (Spring Boot) 91](#_Toc194609695)

[29.6 Algorithme de sélection de lentille 92](#_Toc194609696)

[29.7 Interface Angular 92](#_Toc194609697)

[29.8 Navigation & Sécurité 93](#_Toc194609698)

[29.9 Piste d’amélioration future 93](#_Toc194609699)

[30. Module Summary (Résumé opératoire du patient) 93](#_Toc194609700)

[30.1 Objectif fonctionnel 93](#_Toc194609701)

[30.2 Étapes d’accès 94](#_Toc194609702)

[30.3 Structure Angular du module 94](#_Toc194609703)

[30.4 Backend — Génération du Summary 94](#_Toc194609704)

[30.5 DTO et sérialisation 95](#_Toc194609705)

[30.6 Interface utilisateur Angular 96](#_Toc194609706)

[30.7 Service Angular 96](#_Toc194609707)

[30.8 Sécurité et rôles 97](#_Toc194609708)

[30.9 Problématiques techniques rencontrées 97](#_Toc194609709)

[31. Gestion des utilisateurs et des rôles 97](#_Toc194609710)

[31.1 Objectif 97](#_Toc194609711)

[31.2 Entité User (utilisateur) 98](#_Toc194609712)

[31.3 Enum Role 98](#_Toc194609713)

[31.4 Cycle d’authentification complet 99](#_Toc194609714)

[31.4.1 Inscription (/auth/register) 99](#_Toc194609715)

[31.4.2 Connexion (/auth/authenticate) 99](#_Toc194609716)

[31.4.3 Utilisation des ressources 99](#_Toc194609717)

[31.4.4 Déconnexion (/auth/logout) 99](#_Toc194609718)

[31.5 Table Token et suivi des connexions 100](#_Toc194609719)

[31.6 Gestion frontend des rôles et sessions 100](#_Toc194609720)

[31.7 Vue d’administration des utilisateurs (TODO) 100](#_Toc194609721)

[32. Calcul de la lentille intraoculaire (IOL) 100](#_Toc194609722)

[32.1 Objectif 100](#_Toc194609723)

[32.2 Structure de l’entité Calcul 101](#_Toc194609724)

[32.3 Structure de l’entité Lens 101](#_Toc194609725)

[32.4 Fonctionnalités métier 101](#_Toc194609726)

[32.5 Affichage dans le composant Calculator (Frontend) 102](#_Toc194609727)

[32.6 Logique Backend 102](#_Toc194609728)

[32.7 Cas d’utilisation typique 102](#_Toc194609729)

[32.8 Intégration avec la synthèse opératoire (Summary) 102](#_Toc194609730)

[32.9 Extension possible (Roadmap) 103](#_Toc194609731)

[33. Gestion des antécédents médicaux et ophtalmologiques 103](#_Toc194609732)

[33.1 Objectif général 103](#_Toc194609733)

[33.2 Structure des données 103](#_Toc194609734)

[33.3 Fonctionnement Frontend 104](#_Toc194609735)

[33.4 Traitement côté backend 104](#_Toc194609736)

[33.5 Particularités UX et cas d’usage métier 105](#_Toc194609737)

[33.6 Évolutions possibles 105](#_Toc194609738)

[34. Gestion du côté dominant et de la latéralité 105](#_Toc194609739)

[34.1 Objectif clinique et fonctionnel 105](#_Toc194609740)

[34.2 Données concernées dans l’entité Exam 105](#_Toc194609741)

[34.3 Gestion côté frontend (Angular) 106](#_Toc194609742)

[34.4 Exploitation dans le DTO résumé (SummaryDto) 107](#_Toc194609743)

[34.5 Intégration dans les workflows chirurgicaux 107](#_Toc194609744)

[34.6 Évolutions envisagées 107](#_Toc194609745)

[35. Gestion des implants intraoculaires (IOLs) 108](#_Toc194609746)

[35.1 Contexte médical et finalité logicielle 108](#_Toc194609747)

[35.2 Entités concernées 108](#_Toc194609748)

[35.2.1 Entité Lens.java (implant) 108](#_Toc194609749)

[35.2.2 Enumération LensType 109](#_Toc194609750)

[35.2.3 Entité Calcul.java 109](#_Toc194609751)

[35.3 Relations entre Exam, Calcul et Lens 109](#_Toc194609752)

[35.4 Exploitation dans le backend 109](#_Toc194609753)

[35.5 Exploitation dans le frontend 110](#_Toc194609754)

[35.6 Sécurité et audit 110](#_Toc194609755)

[35.7 Évolutions possibles 111](#_Toc194609756)

[36. Calculs biométriques et formules ophtalmiques 111](#_Toc194609757)

[36.1 Objectifs cliniques et fonctionnels 111](#_Toc194609758)

[36.2 Données biométriques traitées 111](#_Toc194609759)

[36.3 Entité Calcul et prédiction 111](#_Toc194609760)

[36.4 Fonctionnement d’un calcul 112](#_Toc194609761)

[36.5 Sélection du meilleur calcul 113](#_Toc194609762)

[36.6 Services métier associés 113](#_Toc194609763)

[36.7 Interface utilisateur (Angular) 113](#_Toc194609764)

[36.8 Sécurité et audit 114](#_Toc194609765)

[36.9 Perspectives d’évolution 114](#_Toc194609766)

[37. Conclusion 114](#_Toc194609767)

# Introduction

Le projet EyeWeb est né d’un besoin concret exprimé par un médecin ophtalmologue, le docteur François-Xavier Crahay, confronté à une problématique fréquente dans le domaine médical : le manque d’outils informatiques réellement adaptés à la gestion préopératoire des patients. Plus spécifiquement, l’objectif était de concevoir une application web permettant de centraliser l’ensemble des données cliniques et biométriques nécessaires à la préparation des interventions en ophtalmologie, tout en facilitant la collaboration entre le médecin, son assistante, et ses patients.

Ce travail de fin d’études s’inscrit donc dans une double démarche : d’une part, répondre à une demande professionnelle précise en concevant une application fonctionnelle, et d’autre part, démontrer la maîtrise des technologies modernes du développement logiciel au travers d’un projet complet, structuré et évolutif.

Très tôt, il est apparu indispensable de refondre entièrement une version antérieure de l’application pour en améliorer l’architecture, la lisibilité du code, et la cohérence fonctionnelle. Cette réécriture s’est accompagnée d’une analyse rigoureuse des besoins, d’une restructuration complète de la base de données, ainsi que d’une reprise intégrale de l’interface utilisateur.

Le projet EyeWeb repose sur un socle technologique moderne, basé sur Spring Boot (Java 17) côté serveur, et Angular 17 avec PrimeNG côté client. La gestion des données est assurée par PostgreSQL, le tout orchestré en environnement Dockerisé. Une attention particulière a été portée à la sécurité (authentification JWT), à l’ergonomie, ainsi qu’à la modularité du code afin de faciliter l’extension future de la plateforme.

Tout au long du développement, la méthode Agile Scrum a été appliquée : des échanges hebdomadaires ont eu lieu avec le client pour ajuster les priorités, valider les modules livrés, et faire évoluer la plateforme en fonction de ses retours. Cette démarche itérative a permis de faire évoluer l’application en permanence tout en garantissant l’adéquation avec les besoins métiers réels.

Ce document présente l’ensemble de la démarche suivie : de l’analyse initiale à l’implémentation technique, en passant par les choix d’architecture, les enjeux de sécurité, les difficultés rencontrées, et les perspectives d’évolution.

# 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 EyeWeb, tant du côté frontend que backend.

## Backend

Le backend de l’application EyeWeb 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 EyeWebApplication.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 EyeWeb 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 EyeWebApplication.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 EyeWeb. 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 EyeWeb\_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 :

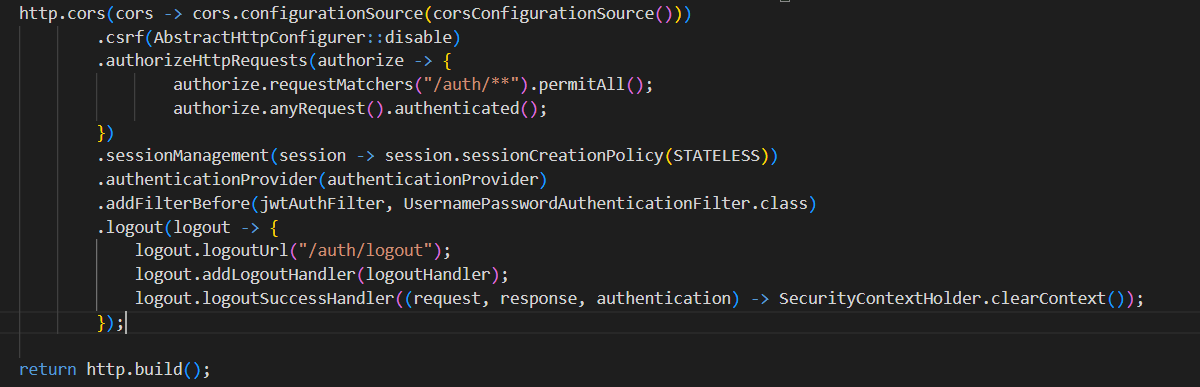
* Authentification forte avec JWT (stateless, sécurisée, portable)
* Séparation nette entre utilisateurs authentifiés et anonymes
* Vérification systématique de chaque requête HTTP entrante
* Gestion dynamique des rôles (ROLE\_ADMIN, ROLE\_MANAGER, ROLE\_CLIENT)
* Révocation possible de tokens et gestion 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

# Architecture générale du projet EyeWeb

L’architecture suit une architecture hexagonale inspirée du modèle MVC étendu, complétée par un service layer pattern et une approche API First. Cette architecture facilite la maintenabilité, et l’évolutivité.

## Organisation modulaire des couches

Le backend est divisé en couches clairement séparées, chacune ayant un rôle précis :

### Controller layer

* Gère les requêtes HTTP entrantes (via @RestController)
* Délègue les traitements métiers à la couche service
* Renvoie des DTO vers le frontend
* controller/

Exemple : AuthenticationController.java, ExamController.java

### Service layer

* Contient toute la logique métier
* Fait le lien entre les contrôleurs REST et les repositories JPA
* Traite les règles de validation, calculs, conditions
* service/, service/impl/

Exemples : AuthenticationServiceImpl.java, ExamServiceImpl.java

### Repository layer

* Accès aux données en base via Spring Data JPA
* Méthodes CRUD automatiques ou personnalisées via @Query
* repository/

Exemples : UserRepository.java, TokenRepository.java, ExamRepository.java

### Entity layer

* Représente les tables PostgreSQL sous forme d’objets Java (avec @Entity)
* Utilise JPA, Hibernate, et annotations de validation
* model/

Exemples : User.java, Exam.java, Patient.java, Calcul.java, Lens.java

### DTOs & Mappers

* Objets de transfert (DTO) exposés à l’extérieur
* Permettent de découpler la base de données des réponses envoyées au frontend
* Mappés avec MapStruct
* dto/, mapper/

Exemples : SummaryDto.java, UserDto.java

## Technologies utilisées par couche

|  |  |
| --- | --- |
| Couche | Technologies principales |
| Controller | Spring Boot, Jackson (JSON), Annotations @RestController |
| Service | Java 17, Spring Beans, @Transactional |
| Repository | Spring Data JPA, @Query, JpaRepository<T, ID> |
| Entity | JPA, Hibernate, PostgreSQL, Lombok |
| DTO / Mapper | MapStruct, Lombok, Validation annotations (@NotNull) |

## Particularités de l’architecture

### Approche API First

Les endpoints sont définis en amont dans les contrôleurs, avec une structure claire par fonctionnalité métier (/auth, /exam, /patient, etc.). Cela permet au frontend de se développer en parallèle, en connaissance de la structure des objets et des routes.

### Architecture RESTful

Chaque ressource possède ses propres routes HTTP bien définies :

* GET /api/exams/{id}
* POST /api/exams
* PUT /api/exams/{id}
* DELETE /api/exams/{id}

Les verbes HTTP sont utilisés conformément aux conventions REST :

* GET : lecture
* POST : création
* PUT : mise à jour
* DELETE : suppression

## Architecture du frontend Angular

L’interface EyeWeb utilise Angular 16, respectant une architecture en composants, avec modules fonctionnels bien structurés, services, et routage dynamique.

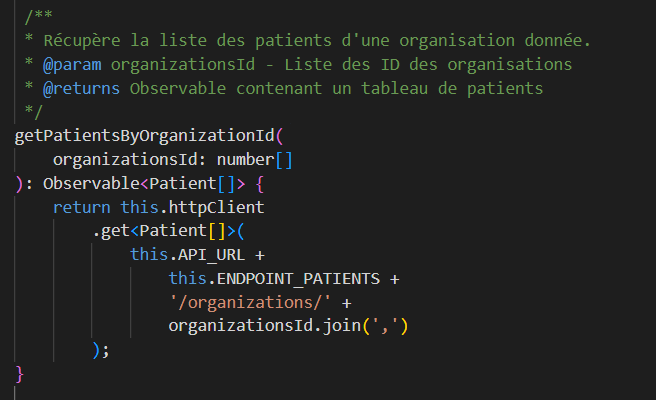
### Structure des fichiers Angular

* src/app/
* components/ → composants métiers (ex: Calculator, Summary, Biometry, Navbar)
* services/ → accès HTTP aux endpoints backend
* models/ → interfaces typescript (Exam.ts, Patient.ts, etc.)
* pages/ → pages principales liées au routing
* shared/ → composants réutilisables, pipes, etc.

### Services Angular

Chaque entité dispose d’un service injectable Angular :

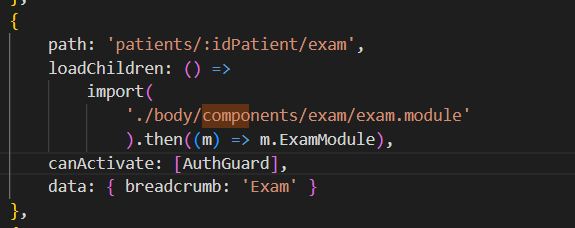
Exemple : PatientService.ts



Ces services utilisent HttpClient et sont injectés dans les composants grâce à @Injectable({ providedIn: 'root' }).

### Routage Angular

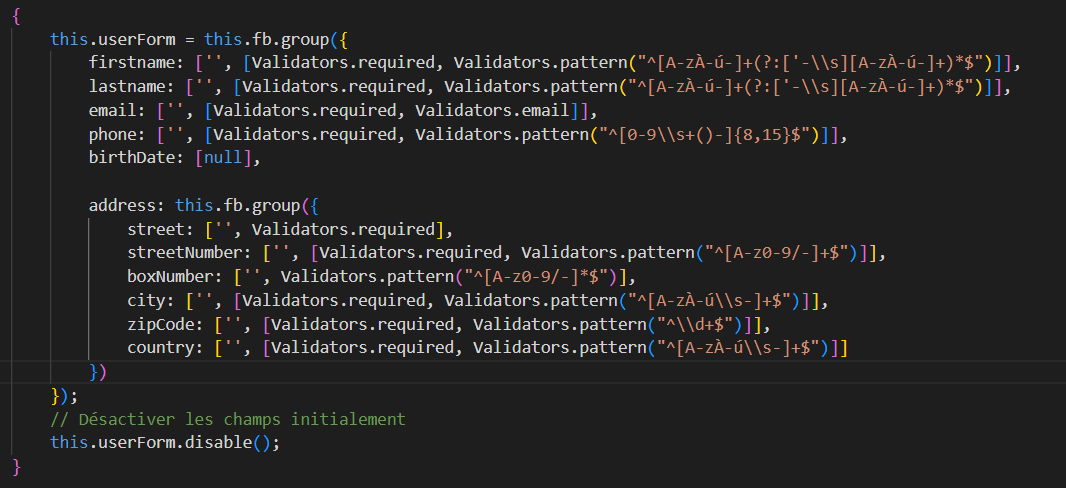
Le fichier app-routing.module.ts déclare les routes principales :



Les routes permettent une navigation dynamique entre les pages, avec des paramètres d’URL (:patientId, etc.) pour charger les bons objets depuis l’API.

### Forms & Reactive Forms

Les formulaires Angular utilisent le module ReactiveFormsModule :



Les validations côté client sont combinées avec les vérifications backend pour plus de robustesse.

## Services transversaux

* Logger intégré à chaque service backend (via SLF4J)
* Filtres globaux : JwtAuthenticationFilter
* Exception handler personnalisé (@ControllerAdvice recommandé)
* Configuration centralisée dans application.yaml et application-dev.yaml

# Frontend : Composants, flux de données, services et expérience utilisateur

L’interface utilisateur de l’application EyeWeb repose sur une architecture modulaire, réactive et orientée composants, implémentée avec Angular 16, TypeScript et SCSS, combinée à PrimeNG pour les composants graphiques interactifs et à RxJS pour la gestion asynchrone des données.

L’approche adoptée suit les meilleures pratiques de conception frontend :

* Séparation des responsabilités
* Réutilisabilité des composants
* Gestion propre des services API.

## Structure générale du frontend Angular

Le frontend est organisé de manière cohérente autour de trois axes :

* Pages (pages) : vues principales mappées aux routes.
* Composants (components) : éléments UI réutilisables ou spécifiques.
* Services (services) : logique d’accès aux données (via HTTP).
* /src/app/ — Racine du projet Angular

|  |  |
| --- | --- |
| Répertoire | Rôle |
| components/ | Composants fonctionnels (tableaux, formulaires, vues de détail, etc.) |
| pages/ | Pages principales routables (ex : home, summary, calculator) |
| services/ | Logique métier côté client et appels API |
| models/ | Interfaces TypeScript des entités (Patient, Exam, Calcul...) |
| shared/ | Composants réutilisables (navbar, modals, pipes...) |
| config/ | Configuration globale et injection de dépendances |

## Composants personnalisés

Chaque composant Angular correspond à une fonctionnalité métier (ex : résumé opératoire, biometry, calcul IOL, gestion patients).

Exemple : SummaryComponent

* Affiche les données d’un patient et de son dernier Exam
* Utilise un p-card PrimeNG pour présenter l'information par bloc
* Affiche les informations patient, chirurgien, IOL choisie, etc.

Exemple : CalculatorComponent

* Utilise des tableaux p-table dynamiques pour afficher les résultats de calcul
* Interaction avec le backend via un ExamService
* Événements utilisateurs déclenchent les recalculs ((click), (change)...)

Style général :

* SCSS modulaire, pas de SCSS global
* styles.scss utilisé pour les variables communes
* Responsive design garanti par PrimeNG + Bootstrap

## Système de routage

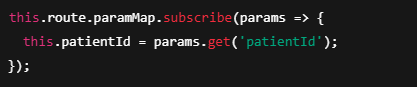
Défini dans app-routing.module.ts, le routage Angular permet de charger dynamiquement les composants en fonction de l’URL.



Le routing dynamique permet de :

* Naviguer vers un patient spécifique via son ID
* Afficher une liste des patients d’une organisation (dashboard.module)
* Séparer les pages par modules

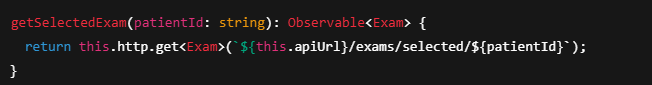
Utilisation combinée avec ActivatedRoute :



## Services Angular & Flux RxJS

Chaque entité possède un Service dédié injecté globalement via providedIn: 'root'.

Exemple : ExamService.ts

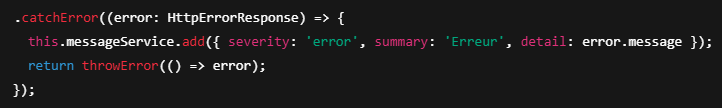


* Utilise HttpClient d’Angular
* Retourne des Observable via RxJS
* Peut être combiné avec des pipe(map(), catchError())

Flux unidirectionnel :

* Le service récupère les données du backend
* Le composant les abonne (subscribe()) et les affiche dans le template

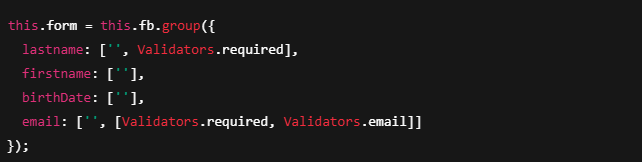
Gestion des erreurs :



## Gestion des formulaires

L’application utilise Reactive Forms pour construire dynamiquement des formulaires complexes, souvent liés à des objets Patient, Exam, Lens, etc.

Exemple : Création patient



* Validation côté client avec Validators
* Contrôle total du formulaire dans le code TypeScript
* Liaison bidirectionnelle avec le modèle ([formControl])

## Bibliothèques UI : PrimeNG

L’interface repose principalement sur PrimeNG, une bibliothèque UI très riche qui facilite l’implémentation d’interfaces dynamiques et professionnelles.

Composants utilisés :

|  |  |
| --- | --- |
| Composant | Fonction |
| p-table | Affichage de listes (Examens, Patients) |
| p-tree | Arborescence des antécédents médicaux |
| p-dialog | Fenêtres modales pour édition/détail |
| p-dropdown | Menus déroulants dynamiques |
| p-toast | Notifications utilisateur |

Les filtres dynamiques et la pagination sont intégrés nativement ([filter]="true" sur les colonnes).

## UI, UX et accessibilité

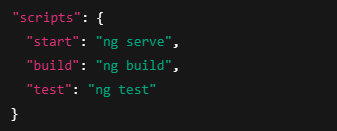
* Dark Mode compatible (si activé dans PrimeNG)
* Rafraîchissement des composants en cas de mise à jour (ngOnInit + ChangeDetectorRef)
* Accessibilité visuelle des composants (écrans chirurgien, résumé, etc.)
* Toasts utilisateurs pour chaque action : suppression, modification, création
* Composants réutilisables (PatientCard, IolSelector, etc.)

## Sécurité dans Angular

* Tous les appels HTTP utilisent des tokens JWT via l’en-tête Authorization
* Intercepteur Angular (HttpInterceptor) configurable pour injecter le token dans chaque requête
* Gestion automatique des erreurs 401 Unauthorized

## Compilation, packaging et configuration

* Compilation via ng build --configuration=production
* Configurations multiples (environnement, API URL) dans environment.ts
* Scripts npm dans package.json :



# Base de données PostgreSQL et ORM (JPA / Hibernate)

La base de données relationnelle PostgreSQL version 15 est le moteur choisi pour stocker l'ensemble des données de l'application. Cette base est conçue selon des principes de normalisation stricte, assurant la cohérence, la réduction de la redondance et l'intégrité référentielle. Elle est accédée à travers Spring Data JPA qui s'appuie sur Hibernate comme fournisseur ORM.

## Structure générale

La structure de la base repose sur les entités principales suivantes :

* Patient
* Exam
* Calcul
* Lens
* User, Token, VerificationCode (pour la sécurité)
* Link, Material, Appointment, Treatment (si applicable)

Toutes ces entités sont liées entre elles selon les cardinalités suivantes :

* Un Patient peut avoir plusieurs Exam
* Un Exam peut avoir plusieurs Calcul
* Un Calcul est associé à une Lens

## Mapping objet-relationnel (ORM)

Chaque entité Java (@Entity) est mappée vers une table PostgreSQL, avec :

* Un identifiant généré automatiquement (@Id + @GeneratedValue)
* Des colonnes annotées avec @Column
* Des relations entre entités définies via :
* @OneToMany, @ManyToOne, @OneToOne
* @JoinColumn (foreign keys explicites)

## Requêtes personnalisées (JPQL)

Certaines opérations spécifiques sont réalisées via des requêtes JPQL, comme :

@Query("SELECT e FROM Exam e JOIN e.links l WHERE l.patient.id = :patientId AND e.selected = true")

Optional<Exam> findSelectedExamByPatientId(@Param("patientId") Long patientId);

## Optimisation :

* Utilisation de @Where(clause = "deleted = FALSE") pour le soft delete
* Indexation sur les colonnes de jointure et de recherche (patient\_id, selected...)
* fetch = FetchType.LAZY sur toutes les relations @OneToMany

# Communication API REST et respect de l’approche API-First

L’application est entièrement pilotée par une API REST conforme aux standards modernes. Cette API est consommée par le frontend Angular.

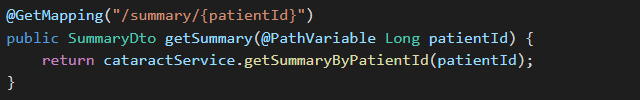
## Standards REST

Les conventions respectées :

* Verbes HTTP :
  + GET /patients, POST /exams, PUT /calculs/{id}, DELETE /exams/{id}
* Statuts HTTP : 200 (OK), 201 (Created), 204 (No Content), 400, 401, 404, 500
* Ressources JSON retournées : camelCase, DTO propres

## Contrôleurs REST (@RestController)

Chaque contrôleur mappe les endpoints d’une entité (PatientController, ExamController...)



## DTOs et Mappers

* Les entités ne sont jamais exposées directement
* Utilisation de SummaryDto, ExamDto, etc.
* Transformation assurée par MapStruct

## Services

Les services assurent la logique métier et orchestrent les appels aux repositories.

# Tests, qualité du code et outils

## Tests unitaires et d’intégration (Backend)

* JUnit 5 : présent dans les dépendances Maven
* @SpringBootTest prêt à l’emploi pour tester les services et contrôleurs
* Utilisation possible de TestContainers pour PostgreSQL (pas encore intégré)

## Tests frontend

* Framework Jasmine + Karma (par défaut dans Angular)
* Fichiers de test \*.spec.ts prévus dans chaque composant
* Possibilité d’ajouter Cypress pour les tests E2E

## Qualité de code

* Linting : ng lint (Angular CLI)
* Typage strict TypeScript (tsconfig.json)
* Fichiers SCSS modulaires (pas de style inline)

## CI/CD

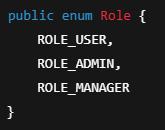
* Aucune intégration continue automatisée n’est déployée dans le projet actuel
* Possibilité d’intégration future avec GitHub Actions ou GitLab CI

# Gestion des rôles et des droits (RBAC - Role-Based Access Control)

La gestion des autorisations dans l'application EyeWeb repose sur un mécanisme robuste de contrôle d'accès basé sur les rôles (RBAC), intégré au cœur de Spring Security et appliqué tant au niveau des endpoints REST que dans la couche de service.

## Définition des rôles

Les rôles définis dans l'application sont généralement modélisés dans une énumération (enum) Role, dont les constantes sont typiquement ROLE\_USER, ROLE\_ADMIN, ROLE\_MANAGER, etc. Chaque utilisateur dans la base de données possède un ou plusieurs rôles qui définissent les permissions d’accès aux différentes ressources applicatives.

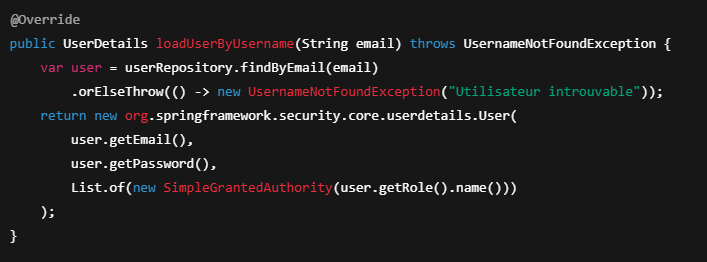


## Stockage et affectation

Les rôles sont stockés dans l’entité User, sous forme de champ Role role;. Lors de l’enregistrement (POST /auth/register), un rôle est automatiquement assigné au nouvel utilisateur (par défaut ROLE\_USER, sauf configuration personnalisée).

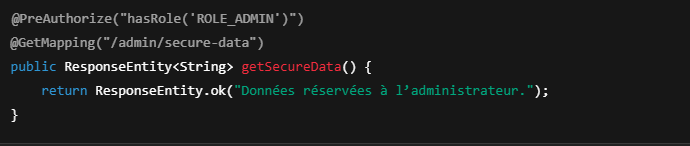
## Chargement via UserDetailsService

La classe UserDetailsServiceImpl implémente UserDetailsService pour intégrer Spring Security. Elle retourne un objet UserDetails contenant les rôles de l’utilisateur, mappés vers des GrantedAuthority.



## Application des droits dans les endpoints

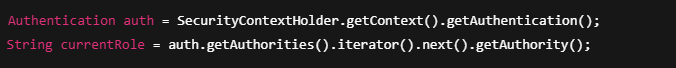
Les contrôleurs REST utilisent des annotations comme @PreAuthorize pour restreindre l’accès :



Ce mécanisme garantit que seuls les utilisateurs authentifiés ayant le bon rôle peuvent accéder aux données sensibles.

## Vérification côté service

Outre les contrôleurs, les services peuvent aussi vérifier les rôles via le SecurityContextHolder :



Ce code permet une vérification manuelle ou une logique conditionnelle plus poussée dans la couche service.

## Avantages de RBAC dans l'application

* Sécurité renforcée : restriction d’accès aux fonctionnalités critiques.
* Extensibilité : possibilité d’ajouter de nouveaux rôles ou permissions.
* Centralisation : toutes les règles d’accès sont centralisées via Spring Security.

# Gestion des erreurs, des performances et des logs

L’application EyeWeb s’appuie sur une gestion rigoureuse des erreurs et une stratégie de journalisation structurée pour assurer stabilité, observabilité et maintenabilité.

## Gestion des erreurs dans le backend

L’API REST gère les erreurs via un mécanisme global basé sur @ControllerAdvice, capturant toutes les exceptions dans une classe personnalisée :

@ControllerAdvice

public class GlobalExceptionHandler {

@ExceptionHandler(ResourceNotFoundException.class)

public ResponseEntity<Object> handleResourceNotFound(ResourceNotFoundException ex) {

return new ResponseEntity<>(new ErrorDto(ex.getMessage()), HttpStatus.NOT\_FOUND);

}

@ExceptionHandler(Exception.class)

public ResponseEntity<Object> handleGlobalException(Exception ex) {

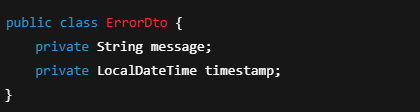
return new ResponseEntity<>(new ErrorDto("Erreur serveur : " + ex.getMessage()), HttpStatus.INTERNAL\_SERVER\_ERROR);

}

}

## Formats de réponse d'erreurs (DTO)

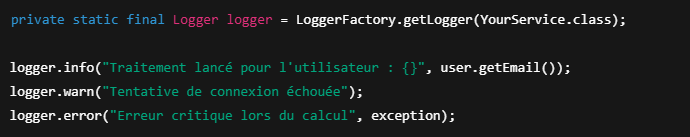
Les erreurs sont renvoyées de manière uniforme grâce à des DTO comme ErrorDto, contenant typiquement :



Cela permet au frontend d’interpréter facilement les erreurs et de les afficher de manière lisible pour l’utilisateur.

## Journalisation avec SLF4J / Logback

L’application utilise SLF4J comme façade de logging, couplée à Logback comme moteur de logs. Cela permet une journalisation centralisée, configurable et performante :



Le fichier application.yaml peut contenir une configuration personnalisée pour les niveaux de logs (TRACE, DEBUG, INFO, WARN, ERROR) selon les environnements (dev/prod).

## Logs de sécurité et d’audit

Toutes les authentifications, connexions, et actions critiques (ex. : suppression d’un patient) sont loggées.

Les accès non autorisés sont interceptés et consignés dans les logs système via les filtres JWT et les gestionnaires d’exception.

## Gestion des performances

Bien que l’application ne comporte pas encore de mécanisme de monitoring avancé, les bonnes pratiques de performances sont respectées :

* Utilisation de @Transactional(readOnly = true) pour les requêtes en lecture seule.
* Pagination et Pageable utilisés dans les endpoints massifs.
* Indexation stratégique dans PostgreSQL (vue dans le document d’analyse).

## Analyse future

L’architecture permet l’ajout de solutions comme :

* Spring Actuator pour les métriques et l’état de l’application.
* ELK Stack (Elasticsearch, Logstash, Kibana) pour l’analyse temps réel des logs.
* Prometheus + Grafana pour les indicateurs applicatifs et système.

# Système de routing Angular et navigation entre composants

Le système de routing Angular constitue la colonne vertébrale de la navigation au sein de l’application EyeWeb, assurant un comportement de type SPA (Single Page Application), fluide et sans rechargement de page.

## Initialisation du routing

Le routing est initialisé dans le fichier app-routing.module.ts, qui contient la configuration centrale des routes. Chaque route est associée à un chemin (path) et à un composant (component).

ts

CopierModifier

const routes: Routes = [

{ path: 'home', component: HomeComponent },

{ path: 'summary/:id', component: SummaryComponent },

{ path: 'calculator', component: CalculatorComponent },

{ path: '', redirectTo: '/home', pathMatch: 'full' },

{ path: '\*\*', component: NotFoundComponent }

];

redirectTo: permet de rediriger la racine (/) vers /home.

:id: indique que l'URL accepte un paramètre dynamique (ex. : /summary/45).

## Composants liés au routing

Chaque composant Angular est lié à une route spécifique :

/home → HomeComponent : écran d’accueil ou de tableau de bord.

/summary/:id → SummaryComponent : page de synthèse d’un patient ou d’un examen.

/calculator → CalculatorComponent : interface de calcul de l’IOL ou de l’implant.

/login → LoginComponent : formulaire de connexion.

/register → RegisterComponent : formulaire d’inscription.

/not-found → NotFoundComponent : page d’erreur 404.

## Navigation entre pages

La navigation dans le code se fait via le service Router d’Angular :

ts

CopierModifier

constructor(private router: Router) {}

goToSummary(id: number) {

this.router.navigate(['/summary', id]);

}

Cela permet une transition de page sans rechargement, tout en mettant à jour l’URL.

## Protection des routes : AuthGuard

Pour sécuriser les routes sensibles (comme /summary, /calculator, etc.), un AuthGuard est utilisé. Celui-ci vérifie si l’utilisateur est authentifié avant de l’autoriser à accéder à une route.

ts

CopierModifier

canActivate(): boolean {

return this.authService.isLoggedIn();

}

Le guard est appliqué comme ceci :

ts

CopierModifier

{ path: 'summary/:id', component: SummaryComponent, canActivate: [AuthGuard] }

## Paramètres de route (ActivatedRoute)

Les composants accèdent aux paramètres de route avec ActivatedRoute :

ts

CopierModifier

constructor(private route: ActivatedRoute) {}

ngOnInit(): void {

const patientId = this.route.snapshot.paramMap.get('id');

}

Cela permet de récupérer dynamiquement l’id du patient ou de l’examen à afficher dans SummaryComponent.

## Routes enfant et modules

L'application est pensée de manière modulaire, avec des modules enfants comme config.module.ts, admin.module.ts, etc. Chaque module peut avoir son propre sous-routing :

ts

CopierModifier

const adminRoutes: Routes = [

{ path: 'users', component: UserManagementComponent },

{ path: 'roles', component: RoleManagementComponent },

];

Ces routes sont chargées via lazy loading dans le AppRoutingModule, optimisant les performances.

ts

CopierModifier

{ path: 'admin', loadChildren: () => import('./admin/admin.module').then(m => m.AdminModule) }

## Navigation par lien (template HTML)

La navigation dans le HTML se fait avec le directive routerLink :

html

CopierModifier

<a [routerLink]="['/summary', patient.id]">Voir résumé</a>

Et la zone centrale de l’interface change dynamiquement avec :

html

CopierModifier

<router-outlet></router-outlet>

C’est ce <router-outlet> qui détermine dynamiquement quel composant afficher dans la page selon l’URL active.

## Gestion des erreurs et redirections

Des routes par défaut sont prévues pour gérer les cas d'erreurs :

Une route avec path: '\*\*' capture les URL inexistantes.

Le NotFoundComponent affiche un message d'erreur lisible.

# Services Angular et gestion des données via RxJS et HTTPClient

La gestion de la logique métier côté frontend dans EyeWeb repose sur les services Angular injectables. Ces services centralisent les appels API, encapsulent la logique métier, et assurent une communication asynchrone avec le backend via HTTP, tout en exploitant la programmation réactive (RxJS).

## Rôle des services dans l'architecture Angular

Dans une architecture Angular modulaire et maintenable, les services jouent un rôle fondamental :

Ils assurent la centralisation de la logique métier et des appels HTTP.

Ils permettent la réutilisation du code à travers différents composants.

Ils permettent une séparation claire des responsabilités : les composants ne sont responsables que de l’affichage, tandis que les services gèrent les données.

Ils permettent une communication fluide entre composants, notamment via des Subjects RxJS.

Chaque service est annoté avec le décorateur @Injectable({ providedIn: 'root' }), ce qui garantit une instance unique partagée dans toute l’application.

## Appels API : HTTPClientModule

Les services utilisent HttpClient, un service natif Angular issu du module HttpClientModule (importé dans AppModule) :

ts

CopierModifier

constructor(private http: HttpClient) {}

Les appels vers le backend sont faits avec des méthodes telles que :

ts

CopierModifier

get<T>(), post<T>(), put<T>(), delete<T>()

Chaque appel est asynchrone et retourne un Observable typé, ce qui permet de souscrire dans le composant et de réagir aux changements de manière réactive.

Exemple d’appel pour récupérer un patient :

ts

CopierModifier

getPatient(id: number): Observable<Patient> {

return this.http.get<Patient>(`${this.apiUrl}/patients/${id}`);

}

## Gestion des Observables et RxJS

La bibliothèque RxJS est au cœur de la gestion de la réactivité et des flux de données.

Types d'observables utilisés :

Observable<T> : données asynchrones simples.

BehaviorSubject<T> : observable avec une valeur par défaut, utilisé pour maintenir un état partagé.

Subject<T> : observable sans valeur initiale, utilisé pour propager un événement.

Exemple typique :

ts

CopierModifier

private currentPatient = new BehaviorSubject<Patient | null>(null);

setCurrentPatient(patient: Patient): void {

this.currentPatient.next(patient);

}

getCurrentPatient(): Observable<Patient | null> {

return this.currentPatient.asObservable();

}

Ce mécanisme permet de partager l’état global entre différents composants sans les lier directement.

## Injection de services et dépendances

Les services Angular sont injectés dans les constructeurs des composants via l’injection de dépendance :

ts

CopierModifier

constructor(private patientService: PatientService) {}

Cela permet une grande testabilité, une découplage fort et une réutilisabilité accrue du code.

## Services métiers disponibles dans EyeWeb

EyeWeb repose sur plusieurs services injectables. En voici quelques exemples significatifs :

| Service | Rôle principal |
| --- | --- |
| PatientService | Récupère, met à jour et supprime les données patient |
| ExamService | Gestion des examens (biométrie, cataracte...) |
| AuthService | Gère l’authentification et la gestion des tokens |
| SummaryService | Appelle l’endpoint backend pour récupérer la synthèse opératoire |
| CalculatorService | Envoie les paramètres pour les calculs IOL |
| LanguageService | Gère la langue de l’interface (i18n personnalisée) |

## Traitement des erreurs globales

Chaque appel API est encapsulé dans une gestion d’erreur standardisée via catchError() :

ts

CopierModifier

getPatient(id: number): Observable<Patient> {

return this.http.get<Patient>(`${this.apiUrl}/patients/${id}`).pipe(

catchError(this.handleError)

);

}

Et le handleError() dans le service :

ts

CopierModifier

private handleError(error: HttpErrorResponse) {

console.error('Une erreur est survenue:', error.message);

return throwError(() => new Error('Erreur API. Veuillez réessayer plus tard.'));

}

Cela permet d’unifier les messages d’erreur côté interface et d’éviter des plantages de l’application.

## Intégration des services dans les composants

Les services sont exploités dans les composants comme suit :

ts

CopierModifier

ngOnInit(): void {

this.patientService.getPatient(this.id).subscribe({

next: patient => this.patient = patient,

error: err => this.errorMessage = err.message

});

}

Ce schéma standard est utilisé dans tous les modules de l’application pour assurer une structure homogène et robuste.

## Services transversaux

En complément, des services transversaux assurent la gestion globale de l’état ou du comportement :

ToastService ou NotificationService : pour afficher les messages de confirmation.

LoadingService : pour afficher un loader global.

StorageService : pour interagir avec localStorage ou sessionStorage.

## Avantages de cette architecture de services

Modularité : chaque service correspond à une entité métier.

Réutilisabilité : les méthodes sont utilisées dans plusieurs composants.

Testabilité : possibilité d’injecter des mocks dans les tests unitaires.

Lisibilité : séparation entre logique métier et interface utilisateur.

Réactivité : adaptation instantanée aux données via RxJS.

# Structure des Composants Angular : Organisation, Interaction et Bonnes Pratiques

Le cœur d’une application Angular repose sur ses composants. Ces derniers sont des blocs fonctionnels représentant les différentes vues et fonctionnalités. Le projet EyeWeb suit une architecture modulaire bien structurée, où chaque composant est responsable d’une unité de présentation spécifique, tout en interagissant avec des services injectés et d’autres composants.

## Anatomie d’un composant Angular

Chaque composant Angular est défini dans un fichier .ts, accompagné de ses fichiers .html (template), .scss (style) et parfois .spec.ts (tests unitaires).

Un composant typique ressemble à ceci :

ts

CopierModifier

@Component({

selector: 'app-patient-summary',

templateUrl: './patient-summary.component.html',

styleUrls: ['./patient-summary.component.scss']

})

export class PatientSummaryComponent implements OnInit {

@Input() patientId!: number;

constructor(private summaryService: SummaryService) {}

ngOnInit(): void {

this.loadSummary();

}

private loadSummary(): void {

this.summaryService.getSummaryByPatientId(this.patientId).subscribe(...);

}

}

Ce modèle est généralisé à travers toute l’application.

## Hiérarchie et découpage des composants

La hiérarchie des composants respecte un découpage logique :

| Catégorie | Exemples | Rôle |
| --- | --- | --- |
| Composants Pages | home, calculator, summary | Représentent les pages principales du routage |
| Composants Métier | patient-summary, exam-detail | Affichent des entités métier spécifiques |
| Composants Techniques | toast, loading, modal-confirm | Fournissent des fonctionnalités transversales |
| Composants Modaux | edit-exam-dialog, create-patient | Sont déclenchés par des événements utilisateur |

## Cycle de vie Angular utilisé

EyeWeb exploite les hooks de cycle de vie Angular afin d’organiser l’exécution du code :

| Hook Angular | Rôle |
| --- | --- |
| ngOnInit() | Initialisation, appels aux services, récupération des données |
| ngOnDestroy() | Nettoyage des subscriptions RxJS, timer, etc. |
| ngAfterViewInit() | Manipulation DOM (rarement utilisée ici, mais supportée) |

Ces hooks sont utilisés dans chaque composant critique, notamment ceux du calcul ou du résumé opératoire.

## Communication entre composants

Angular propose plusieurs moyens pour faire interagir les composants :

a) @Input et @Output

@Input() permet de passer des données d’un parent vers un enfant.

@Output() combiné à EventEmitter permet de remonter un événement.

ts

CopierModifier

@Input() selectedExam!: Exam;

@Output() examValidated = new EventEmitter<number>();

b) Services avec RxJS (BehaviorSubject)

Pour les données partagées à plusieurs niveaux (par exemple : patient sélectionné), on utilise les BehaviorSubject exposés dans les services.

## Modularité des composants

L'application Angular est divisée en modules fonctionnels (feature modules) regroupant des composants par domaine métier, avec leur propre routing :

calculator.module.ts

summary.module.ts

patient.module.ts

shared.module.ts

Chaque module importe ses composants, ses services, ses routes, ses dépendances UI (PrimeNG, FormsModule…) de manière autonome.

## Chargement dynamique de composants

Certains composants (modaux, dialogues, notifications) sont créés dynamiquement via Angular CDK ou PrimeNG :

ts

CopierModifier

this.dialogService.open(EditExamDialogComponent, {

data: { examId: selectedId }

});

Cela offre une flexibilité UX avancée.

## Interaction avec l’interface (UI)

Les composants interagissent avec les éléments UI via :

Formulaires réactifs (FormGroup, FormControl)

Directives structurelles (\*ngIf, \*ngFor, ngSwitch)

Binding de propriétés ([disabled], [value])

Binding d’événements ((click), (input), (change))

L’approche repose sur un modèle de données unidirectionnel et immutable, en conformité avec les bonnes pratiques Angular.

## Accessibilité et Responsiveness

Les composants utilisent des pratiques favorisant l’accessibilité (aria-label, tabindex, focus trap) et un rendu responsive via :

PrimeNG

Bootstrap (via SCSS personnalisés)

Layouts flexibles (p-grid, p-col, etc.)

## Tests des composants (structure)

Chaque composant peut être testé via Jasmine + Karma, comme défini dans tsconfig.spec.json.

Fichiers .spec.ts permettent de tester :

Création du composant

Interaction avec les services mockés

Affichage de données

Réactions aux événements

## 1Cas d’usage concrets dans EyeWeb

| Composant | Particularité |
| --- | --- |
| summary.component.ts | Affiche un résumé opératoire complet, connecté au backend en temps réel |
| calculator.component.ts | Envoie des paramètres biométriques à l’API et affiche le résultat |
| patient-detail.component | Formulaire réactif pour modifier les infos patient avec validations dynamiques |
| header.component.ts | Gère l’affichage du menu, langue, logout, responsive avec mobile toggle |

# Routing et navigation dans EyeWeb

La navigation au sein d’une application SPA (Single Page Application) est un élément fondamental pour offrir une expérience fluide à l’utilisateur. Dans le projet EyeWeb, cette navigation est assurée par le module Angular Router, qui permet d’associer des URL spécifiques à des composants précis. Ce chapitre détaille l’implémentation, l’organisation et les particularités du routing dans le projet.

## Objectifs du système de routing

Le système de routing dans EyeWeb remplit plusieurs fonctions essentielles :

Permettre une navigation entre les vues de manière fluide et rapide.

Charger dynamiquement les composants sans recharger la page.

Sécuriser les routes sensibles par des gardes (AuthGuard).

Passer des paramètres d’URL aux composants (ex: patientId, examId, etc.).

Structurer les modules fonctionnels pour le lazy loading.

## Structure du routing principal

Le point d’entrée du routing est défini dans le fichier app-routing.module.ts. Celui-ci centralise toutes les routes publiques et protégées de l’application :

ts

Copier

Modifier

const routes: Routes = [

{ path: '', redirectTo: 'home', pathMatch: 'full' },

{

path: 'home',

component: HomeComponent

},

{

path: 'calculator',

component: CalculatorComponent,

canActivate: [AuthGuard]

},

{

path: 'summary/:patientId',

component: SummaryComponent,

canActivate: [AuthGuard]

},

{

path: 'login',

component: LoginComponent

},

{

path: '\*\*',

component: NotFoundComponent

}

];

## Lazy loading des modules

Afin d’optimiser les performances et de respecter les principes d’architecture modulaire, EyeWeb exploite le lazy loading pour charger uniquement les modules nécessaires.

Exemple d’import d’un module en lazy loading :

ts

Copier

Modifier

{

path: 'patients',

loadChildren: () => import('./modules/patient/patient.module').then(m => m.PatientModule),

canActivate: [AuthGuard]

}

Ce découplage permet de réduire le bundle initial, et de faciliter la maintenance de l’application.

## Sécurisation des routes avec AuthGuard

EyeWeb intègre une garde personnalisée (AuthGuard) afin de protéger les routes sensibles :

ts

Copier

Modifier

@Injectable({ providedIn: 'root' })

export class AuthGuard implements CanActivate {

constructor(private authService: AuthService, private router: Router) {}

canActivate(): boolean {

if (this.authService.isAuthenticated()) {

return true;

}

this.router.navigate(['/login']);

return false;

}

}

Les routes comme /calculator, /summary, /patients sont donc accessibles uniquement aux utilisateurs authentifiés via JWT.

## Passage de paramètres dans l’URL

Le système de routing permet de transmettre des informations à travers l’URL. C’est notamment le cas dans :

ts

Copier

Modifier

{ path: 'summary/:patientId', component: SummaryComponent }

Dans le composant SummaryComponent, le paramètre est récupéré via ActivatedRoute :

ts

Copier

Modifier

this.route.paramMap.subscribe(params => {

const id = params.get('patientId');

this.loadSummary(parseInt(id!, 10));

});

Cela permet de naviguer dynamiquement vers une fiche patient ou un examen spécifique.

## Navigation dans le code

La navigation peut aussi être déclenchée programmétiquement à l’aide du Router :

ts

Copier

Modifier

this.router.navigate(['/summary', patient.id]);

Cela permet de rediriger l’utilisateur après un formulaire, une sélection, ou une action.

## Liens dans les templates (HTML)

Dans les templates HTML, la directive routerLink est utilisée :

html

Copier

Modifier

<a [routerLink]="['/calculator']">Calculateur</a>

<a [routerLink]="['/summary', selectedPatient.id]">Résumé</a>

Le routerLinkActive est utilisé pour ajouter dynamiquement des classes CSS aux liens actifs.

## Redirections et pages 404

Le système gère également les redirections et les routes non trouvées :

ts

Copier

Modifier

{ path: '', redirectTo: 'home', pathMatch: 'full' }

{ path: '\*\*', component: NotFoundComponent }

La page NotFoundComponent permet de guider l’utilisateur en cas d’erreur de navigation.

15.9 Routage par modules fonctionnels

Chaque module Angular possède son propre sous-routing :

Exemple : patient-routing.module.ts

ts

Copier

Modifier

const routes: Routes = [

{

path: '',

component: PatientListComponent

},

{

path: ':id',

component: PatientDetailComponent

}

];

Cela permet un encapsulation logique de la navigation dans chaque domaine métier.

## Synchronisation avec le backend

Le routage Angular interagit directement avec le backend via les services :

Lors d’un accès à /summary/:patientId, le composant SummaryComponent appelle l’API GET /api/summary/:id.

L’ID est donc synchronisé entre le frontend (routeur Angular) et le backend (contrôleur Spring Boot).

# Services Angular et consommation d’API REST

Dans une application moderne de type SPA, la communication entre le frontend Angular et le backend Spring Boot repose sur un ensemble de services typés et structurés. Ces services assurent la couche d’interaction réseau, encapsulent les appels aux API REST, et standardisent les traitements des données reçues ou envoyées.

## Rôle des services Angular

Les services Angular ont plusieurs fonctions essentielles dans le projet EyeWeb :

Centraliser les appels HTTP vers l’API.

Fournir une abstraction métier réutilisable dans plusieurs composants.

Gérer les erreurs et les réponses du serveur.

Appliquer des transformations ou des logiques (ex: formatage, tri, mapping).

Garantir la typage strict des données manipulées (via interfaces ou classes).

## Structure des services dans le projet

Les services Angular sont généralement localisés dans un répertoire dédié, tel que :

css

Copier

Modifier

/src/app/services/

├── patient.service.ts

├── exam.service.ts

├── auth.service.ts

├── calculator.service.ts

└── summary.service.ts

Chaque service correspond à une entité métier ou une fonctionnalité transverse (comme l’authentification ou le calcul).

## Exemple de service standardisé

Exemple : patient.service.ts

ts

Copier

Modifier

@Injectable({ providedIn: 'root' })

export class PatientService {

private apiUrl = environment.apiBaseUrl + '/patients';

constructor(private http: HttpClient) {}

getAll(): Observable<Patient[]> {

return this.http.get<Patient[]>(this.apiUrl);

}

getById(id: number): Observable<Patient> {

return this.http.get<Patient>(`${this.apiUrl}/${id}`);

}

create(patient: Patient): Observable<Patient> {

return this.http.post<Patient>(this.apiUrl, patient);

}

update(id: number, patient: Patient): Observable<Patient> {

return this.http.put<Patient>(`${this.apiUrl}/${id}`, patient);

}

delete(id: number): Observable<void> {

return this.http.delete<void>(`${this.apiUrl}/${id}`);

}

}

Ce service respecte le principe CRUD (Create, Read, Update, Delete) via des appels typés au HttpClient.

## Sécurité des requêtes HTTP

Le service d’authentification ajoute automatiquement le token JWT à chaque requête :

ts

Copier

Modifier

intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

const jwt = this.authService.getToken();

if (jwt) {

const cloned = req.clone({

headers: req.headers.set('Authorization', 'Bearer ' + jwt)

});

return next.handle(cloned);

}

return next.handle(req);

}

Ce HttpInterceptor est central dans la chaîne de communication et garantit que seules les requêtes authentifiées sont transmises.

## Utilisation dans les composants

Les services sont injectés dans les composants via le constructeur :

ts

Copier

Modifier

constructor(private patientService: PatientService) {}

ngOnInit(): void {

this.patientService.getAll().subscribe(data => {

this.patients = data;

});

}

Cette injection de dépendance permet un découplage complet entre l’interface utilisateur et la logique métier.

## Gestion centralisée des erreurs

Le projet EyeWeb intègre une stratégie de gestion des erreurs au niveau des services :

ts

Copier

Modifier

catchError((error: HttpErrorResponse) => {

console.error('Erreur API :', error.message);

return throwError(() => new Error('Erreur serveur'));

});

Cela permet de remonter les erreurs au composant, ou de les journaliser dans un service de log.

## Appels conditionnels et typage avancé

Certains services utilisent des conditions, des transformations, ou des types génériques :

ts

Copier

Modifier

search(term: string): Observable<Patient[]> {

return this.http.get<Patient[]>(`${this.apiUrl}/search`, {

params: new HttpParams().set('term', term)

});

}

Ou encore l’usage de BehaviorSubject pour stocker des états :

ts

Copier

Modifier

private patientSelected = new BehaviorSubject<Patient | null>(null);

selected$ = this.patientSelected.asObservable();

select(patient: Patient) {

this.patientSelected.next(patient);

}

## Appels spécifiques : calcul, résumé, auth

Chaque module métier dispose d’un service spécialisé :

calculator.service.ts : envoie les données biométriques au backend, récupère les IOL calculées.

summary.service.ts : appelle l’API /summary/{patientId} et retourne un SummaryDto complet.

auth.service.ts : s’occupe de la connexion, inscription, logout, gestion du JWT.

## Séparation des couches et bonnes pratiques

Le projet respecte la séparation des responsabilités :

Les services font les appels réseau.

Les composants affichent les données.

Les modèles (interfaces) décrivent les structures (Patient, Exam, Calcul, etc.).

Les routes déterminent la navigation.

Cette architecture permet une réutilisabilité et une testabilité maximale.

# Composants, interfaces et gestion des états

Les composants représentent le cœur de l’interface utilisateur d’une application Angular. Dans le projet EyeWeb, chaque module fonctionnel (Patient, Calcul, Résumé opératoire, Authentification...) repose sur un ou plusieurs composants organisés, fortement typés, connectés au backend et entièrement intégrés dans une architecture modulaire.

## Organisation des composants Angular

L’architecture du frontend est modulaire et suit une organisation par domaines fonctionnels. Chaque module contient :

Un composant principal (liste, tableau, formulaire, etc.)

Un service associé (pour les appels HTTP)

Un module de déclaration (ex: patient.module.ts)

Un routing interne (ex: patient-routing.module.ts)

Des modèles typés (interfaces TypeScript)

Exemple :

css

Copier

Modifier

/src/app/

├── auth/

│ ├── login.component.ts

│ ├── register.component.ts

│ └── auth.service.ts

├── patient/

│ ├── patient-list.component.ts

│ ├── patient-form.component.ts

│ ├── patient.model.ts

│ ├── patient.service.ts

│ └── patient.module.ts

├── exam/

├── calculator/

└── summary/

## Structure d’un composant Angular

Un composant Angular repose sur trois fichiers principaux :

.component.ts → logique TypeScript (classe, données, appels services)

.component.html → gabarit HTML de rendu

.component.scss → style visuel localisé au composant

Exemple : summary.component.ts

ts

Copier

Modifier

@Component({

selector: 'app-summary',

templateUrl: './summary.component.html',

styleUrls: ['./summary.component.scss']

})

export class SummaryComponent implements OnInit {

summary: SummaryDto | null = null;

constructor(private summaryService: SummaryService, private route: ActivatedRoute) {}

ngOnInit(): void {

const patientId = this.route.snapshot.params['id'];

this.summaryService.getSummary(patientId).subscribe(res => this.summary = res);

}

}

## Interfaces TypeScript fortement typées

Chaque entité métier du backend (Patient, Calcul, Exam...) possède une interface TypeScript correspondante. Ces interfaces assurent la cohérence entre frontend et backend.

Exemple : Patient.ts

ts

Copier

Modifier

export interface Patient {

id: number;

lastname: string;

firstname: string;

birthDate: string;

job?: string;

email: string;

}

Cela permet :

Le typage strict dans les services

Le binding précis dans les templates HTML

Une lisibilité et maintenabilité accrue

## Liaison de données et formulaire réactif

L’application utilise principalement des formulaires réactifs (FormGroup, FormControl) pour le login, l’inscription, la modification des patients ou examens.

Exemple dans login.component.ts :

ts

Copier

Modifier

this.loginForm = this.fb.group({

email: ['', [Validators.required, Validators.email]],

password: ['', Validators.required]

});

Dans le HTML associé :

html

Copier

Modifier

<form [formGroup]="loginForm" (ngSubmit)="onSubmit()">

<input formControlName="email" type="email">

<input formControlName="password" type="password">

</form>

## Rendu conditionnel et composants dynamiques

Certains composants comme summary.component.ts affichent des sections conditionnelles selon les champs du SummaryDto.

Exemple :

html

Copier

Modifier

<div \*ngIf="summary?.surgeryType">

Type de chirurgie : {{ summary.surgeryType }}

</div>

De plus, des composants comme les tableaux (p-table) ou arbres (p-tree) de PrimeNG sont instanciés de manière dynamique selon les données reçues.

## Communication entre composants

L'application utilise :

@Input() pour envoyer des données à un composant enfant.

@Output() avec EventEmitter pour remonter des événements au parent.

BehaviorSubject dans les services pour diffuser les états globaux.

Exemple :

ts

Copier

Modifier

@Output() userSelected = new EventEmitter<Patient>();

selectUser(user: Patient) {

this.userSelected.emit(user);

}

## État local et global

L’application gère les états locaux via des attributs internes aux composants (patients, selectedExam, summary, etc.) et états globaux partagés via des BehaviorSubject.

Exemple dans calculator.service.ts :

ts

Copier

Modifier

private iolResult = new BehaviorSubject<CalculationResult | null>(null);

iolResult$ = this.iolResult.asObservable();

setIolResult(result: CalculationResult) {

this.iolResult.next(result);

}

## Gestion des erreurs dans les composants

Les erreurs d’appel API ou de formulaire sont capturées dans les composants :

ts

Copier

Modifier

this.authService.login(this.loginForm.value).subscribe({

next: (res) => this.router.navigate(['/home']),

error: (err) => this.errorMessage = 'Erreur d’authentification'

});

Cela permet d’afficher une alerte personnalisée à l’utilisateur :

html

Copier

Modifier

<div class="alert alert-danger" \*ngIf="errorMessage">

{{ errorMessage }}

</div>

## Intégration visuelle et design

Chaque composant est stylisé avec SCSS local et utilise PrimeNG pour les éléments UI :

Boutons (p-button)

Tableaux (p-table)

Arbres (p-tree)

Modales (p-dialog)

Onglets (p-tabView)

Icônes (PrimeIcons)

Les composants sont donc modulaires, réactifs, accessibles et uniformes.

## Résumé

L’ensemble des composants Angular de EyeWeb suivent une architecture claire et typée :

Intégration étroite avec les services backend

Utilisation avancée des modèles et des formulaires

Séparation des responsabilités

Réutilisabilité et testabilité

Ils constituent le noyau fonctionnel de l’interface utilisateur, entièrement connectée au backend sécurisé.

# Routing, Navigation et Modules Angular

Dans une application web moderne basée sur Angular, la gestion de la navigation entre les différentes pages ou composants repose sur un système de routing centralisé. Ce système est structuré autour d’un RouterModule configuré au niveau racine, puis subdivisé au sein de chaque module métier (Patient, Auth, Exam, etc.).

EyeWeb exploite ce mécanisme de manière modulaire et sécurisée, en respectant les principes de l’architecture SPA (Single Page Application).

## Configuration du Routing principal (app-routing.module.ts)

Le fichier app-routing.module.ts constitue l’entrée principale du router Angular. Il définit les chemins (routes) accessibles dans l’application, les composants associés et les éventuelles règles de protection d’accès.

Exemple typique :

ts

Copier

Modifier

const routes: Routes = [

{ path: '', redirectTo: '/home', pathMatch: 'full' },

{

path: 'auth',

loadChildren: () => import('./auth/auth.module').then(m => m.AuthModule)

},

{

path: 'patient',

loadChildren: () => import('./patient/patient.module').then(m => m.PatientModule),

canActivate: [AuthGuard]

},

{

path: 'summary',

loadChildren: () => import('./summary/summary.module').then(m => m.SummaryModule),

canActivate: [AuthGuard]

},

{ path: '\*\*', component: NotFoundComponent }

];

🔹 Utilisation de loadChildren : permet le lazy-loading des modules.

🔹 Mise en place d’une protection des routes par le biais d’un AuthGuard.

🔹 Gestion des routes inconnues avec un composant NotFoundComponent.

## Modules Angular et découpage fonctionnel

EyeWeb repose sur une structure fortement modulaire, chaque entité ou domaine métier dispose de son propre module Angular.

Exemples :

patient.module.ts : pour la gestion des patients

exam.module.ts : pour les examens ophtalmologiques

calculator.module.ts : pour les calculs d’implants intraoculaires

summary.module.ts : pour le résumé opératoire

auth.module.ts : pour l’authentification, l’inscription, le logout

Chaque module contient :

Un \*.module.ts pour les déclarations et imports

Un \*.routing.module.ts pour le routing interne

Des composants spécifiques

Un service métier (ex: ExamService, PatientService)

## Lazy Loading et séparation des responsabilités

Grâce au lazy-loading, les modules ne sont chargés qu’en cas de besoin (au moment de la navigation), ce qui permet :

Une optimisation du temps de chargement initial

Une meilleure gestion de la mémoire

Une séparation claire des responsabilités

Exemple : le module calculator n’est chargé que lorsque l’utilisateur navigue vers /calculator.

## Gestion de la navigation (Router, ActivatedRoute)

Dans les composants, la navigation est réalisée via le service Router. Les paramètres d’URL sont extraits via ActivatedRoute.

Exemple d’accès à un patient spécifique :

ts

Copier

Modifier

constructor(private route: ActivatedRoute) {}

ngOnInit(): void {

const patientId = this.route.snapshot.paramMap.get('id');

this.patientService.getPatientById(patientId).subscribe(...);

}

Et pour naviguer vers une autre page :

ts

Copier

Modifier

this.router.navigate(['/summary', patient.id]);

18.5 Utilisation des paramètres de routes

Certains modules utilisent des routes paramétrées. Exemple :

ts

Copier

Modifier

{ path: 'summary/:id', component: SummaryComponent }

Cela permet de récupérer dynamiquement un identifiant dans l’URL.

## Routage conditionnel et redirections

Certaines routes redirigent automatiquement vers une autre route :

ts

Copier

Modifier

{ path: '', redirectTo: 'auth/login', pathMatch: 'full' }

Cela est utilisé notamment au lancement de l’application ou à la suite d’une déconnexion.

## AuthGuard et accès sécurisé

Le AuthGuard vérifie si l’utilisateur est connecté et possède un token valide avant d’accéder à une route protégée.

Extrait simplifié :

ts

Copier

Modifier

@Injectable()

export class AuthGuard implements CanActivate {

constructor(private authService: AuthService, private router: Router) {}

canActivate(): boolean {

if (this.authService.isAuthenticated()) {

return true;

}

this.router.navigate(['/auth/login']);

return false;

}

}

Ce guard est attaché aux routes sensibles dans le app-routing.module.ts.

## Routing dans les modules internes

Chaque module dispose de son propre fichier de routing, ex. summary-routing.module.ts :

ts

Copier

Modifier

const routes: Routes = [

{ path: '', component: SummaryComponent }

];

Cela permet une gestion locale et autonome du routing.

## Animation de transitions (optionnel)

Des animations de transition entre composants peuvent être ajoutées via le module @angular/animations. Cela améliore l’ergonomie.

## Navigation dans le HTML

Dans les templates, on utilise [routerLink] pour la navigation :

html

Copier

Modifier

<a [routerLink]="['/patient']">Liste des patients</a>

Pour appliquer un style actif :

html

Copier

Modifier

<a routerLink="/summary" routerLinkActive="active">Résumé</a>

## Résumé

Le système de routing dans EyeWeb :

Est centralisé au niveau de app-routing.module.ts

Est délégué aux modules secondaires pour un découplage optimal

Est protégé via AuthGuard et le système JWT

Exploite le lazy-loading pour optimiser la performance

Est intégré dans la navigation utilisateur grâce à routerLink et Router

# Services, Communication HTTP & Gestion des Erreurs dans Angular

L’un des principes fondamentaux d’une architecture Angular bien structurée repose sur la séparation des responsabilités entre les composants de présentation et la logique métier. Dans EyeWeb, cette séparation est assurée par des services dédiés, généralement associés à une entité métier (ex: PatientService, ExamService, SummaryService, etc.), qui encapsulent toute la communication HTTP avec le backend Spring Boot. Ces services centralisent les appels API REST, gèrent la transformation des données, le traitement des erreurs, et parfois le cache local.

## Architecture Générale des Services Angular

Chaque service Angular est défini comme une classe injectable avec le décorateur @Injectable({ providedIn: 'root' }), ce qui garantit qu’une instance unique (singleton) est partagée dans toute l’application.

Exemple d’en-tête type :

ts

Copier

Modifier

@Injectable({

providedIn: 'root'

})

export class PatientService {

constructor(private http: HttpClient) {}

}

🔹 Les services sont injectés dans les composants via le constructeur.

🔹 Ils encapsulent toute la logique métier liée à une entité (CRUD, recherche, traitement).

🔹 Ils permettent de déléguer la responsabilité au bon niveau : les composants se contentent d'appeler les méthodes métier.

## Communication HTTP : le rôle du HttpClient

L’API HttpClient, fournie par le module HttpClientModule d’Angular, est utilisée dans tous les services pour interagir avec le backend.

Les méthodes principales utilisées sont :

get<T>() : pour récupérer des ressources

post<T>() : pour créer une nouvelle ressource

put<T>() : pour mettre à jour

delete<T>() : pour supprimer

Exemple concret :

ts

Copier

Modifier

getPatientById(id: number): Observable<Patient> {

return this.http.get<Patient>(`${this.apiUrl}/patients/${id}`);

}

Dans EyeWeb, tous les appels sont typés avec les modèles TypeScript (Patient, Exam, SummaryDto, etc.), ce qui permet une intégration forte avec le typage Angular.

## Gestion des Observables avec RxJS

Tous les appels HTTP retournent des Observables (Observable<T>), conformément à la philosophie réactive d’Angular.

Cela permet :

Une gestion asynchrone efficace

La composition de flux (mergeMap, switchMap, catchError)

Une intégration fluide avec les formulaires réactifs et les composants dynamiques

Dans les composants, on s’abonne aux observables :

ts

Copier

Modifier

this.patientService.getPatientById(id).subscribe({

next: data => this.patient = data,

error: err => this.handleError(err)

});

## Gestion Centralisée des Erreurs

Une stratégie de gestion d’erreurs est cruciale pour garantir une application robuste.

Dans EyeWeb, chaque service implémente un pattern catchError combiné à throwError :

ts

Copier

Modifier

getAllPatients(): Observable<Patient[]> {

return this.http.get<Patient[]>(`${this.apiUrl}/patients`)

.pipe(

catchError(this.handleError)

);

}

private handleError(error: HttpErrorResponse) {

if (error.status === 0) {

console.error('Erreur côté client ou réseau', error);

} else {

console.error(`Backend retourne code ${error.status}, corps:`, error.error);

}

return throwError(() => new Error('Erreur de communication avec le serveur.'));

}

🔸 Cette logique permet :

D’uniformiser le traitement des erreurs API

De remonter des messages clairs au composant

De logger les erreurs à terme dans une solution externe (Sentry, LogRocket...)

## Sécurisation des requêtes HTTP avec JWT

Toutes les requêtes HTTP doivent être authentifiées via un token JWT. Cela est géré via un HttpInterceptor, qui intercepte chaque requête sortante pour y injecter automatiquement le header :

ts

Copier

Modifier

Authorization: Bearer <token>

Exemple typique de l’interceptor (déjà présent dans EyeWeb) :

ts

Copier

Modifier

intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

const token = this.authService.getToken();

if (token) {

req = req.clone({

setHeaders: {

Authorization: `Bearer ${token}`

}

});

}

return next.handle(req);

}

Cette approche garantit que toutes les requêtes REST vers le backend sécurisé sont protégées par une authentification.

## Services spécialisés et logique métier

Chaque entité dispose de son propre service métier :

Entité Service Angular Exemple de méthodes

Patient PatientService getAll(), getById(), create(), update()

Exam ExamService getByPatient(), create(), update(), delete()

Calcul CalculatorService calculateIOL(), getBestOptions()

Résumé SummaryService getSummaryByPatientId()

Authentification AuthService login(), logout(), register()

## Détection d’erreurs frontend vs backend

Les erreurs peuvent venir :

Du serveur (erreur 500, 403, 404)

Du réseau (offline)

Du frontend (mauvais formulaire, données mal formatées)

La gestion via catchError permet de capturer tout cela proprement, tandis que le service d'authentification peut rediriger vers /auth/login si une erreur 401 est détectée.

## Utilisation dans les composants

Les composants se contentent d’appeler les méthodes des services, souvent via ngOnInit() :

ts

Copier

Modifier

ngOnInit(): void {

this.examService.getByPatientId(this.patientId).subscribe({

next: data => this.exams = data,

error: err => this.errorMessage = err.message

});

}

Cette approche respecte les principes SOLID : notamment le principe de Responsabilité Unique, puisque les composants n’ont aucune logique d’appel réseau.

## Évolution possible vers une couche de "store"

À terme, une évolution naturelle consisterait à centraliser l’état via une solution comme NgRx ou Akita, permettant :

Un store unique contenant l’état global

Des effets RxJS pour gérer les appels asynchrones

Une architecture unidirectionnelle plus solide

Mais pour le moment, l’approche par services suffit pour maintenir une architecture claire et fonctionnelle.

## Résumé

Chaque entité métier est associée à un service Angular typé

Les services utilisent HttpClient pour interagir avec l’API backend

Les erreurs sont centralisées, loggées, et transmises de manière uniforme

Les tokens JWT sont injectés automatiquement via un HttpInterceptor

Les composants restent dépourvus de logique réseau, favorisant la maintenance

# Gestion des Formulaires Réactifs et Validation dans Angular

Dans une application d’entreprise moderne telle qu’EyeWeb, la saisie de données constitue une opération centrale. Pour garantir la fiabilité des entrées utilisateur, Angular offre deux approches pour manipuler les formulaires : les formulaires pilotés par le template (template-driven) et les formulaires réactifs (reactive forms). EyeWeb adopte la seconde, réputée pour sa robustesse, sa modularité, et son adaptabilité dans les interfaces complexes.

## Principe des Formulaires Réactifs

Les formulaires réactifs sont construits via des objets JavaScript (FormGroup, FormControl, FormArray) qui modélisent de manière explicite la structure du formulaire dans le composant TypeScript.

Cela permet une maîtrise programmatique totale sur :

L’état des champs (valide, touché, sale…)

Les erreurs

Les valeurs

Les règles de validation dynamiques

## Mise en place dans EyeWeb

L’infrastructure repose sur l’importation du module ReactiveFormsModule dans le AppModule ou le AppConfigModule.

ts

Copier

Modifier

import { ReactiveFormsModule } from '@angular/forms';

@NgModule({

imports: [

ReactiveFormsModule,

...

]

})

export class AppConfigModule {}

## Création et structure d’un formulaire

Prenons l’exemple d’un formulaire de création de patient dans PatientFormComponent.

ts

Copier

Modifier

form: FormGroup = this.fb.group({

firstname: ['', [Validators.required]],

lastname: ['', [Validators.required]],

email: ['', [Validators.required, Validators.email]],

birthDate: ['', Validators.required],

job: ['']

});

📌 Chaque champ du formulaire est associé à un FormControl, et peut recevoir un ou plusieurs validators synchrones ou asynchrones.

Le constructeur utilise le FormBuilder (this.fb) pour faciliter la création de FormGroup.

## Validation des champs

### Validateurs Intégrés Angular

Angular fournit des validateurs natifs :

Validators.required : champ obligatoire

Validators.email : format e-mail

Validators.minLength(x) / Validators.maxLength(x)

Validators.pattern() : regex personnalisée

Exemple :

ts

Copier

Modifier

email: ['', [Validators.required, Validators.email]]

### Validateurs personnalisés

Dans EyeWeb, des validateurs spécifiques peuvent être ajoutés :

ts

Copier

Modifier

static emailDomainValidator(control: AbstractControl): ValidationErrors | null {

const email = control.value;

return email && email.endsWith('@EyeWeb.com') ? null : { domainInvalid: true };

}

Puis utilisé dans le form :

ts

Copier

Modifier

email: ['', [Validators.required, emailDomainValidator]]

## Affichage des messages d’erreur

Dans le template HTML :

html

Copier

Modifier

<input type="text" formControlName="firstname">

<div \*ngIf="form.get('firstname')?.invalid && form.get('firstname')?.touched">

<small class="text-danger" \*ngIf="form.get('firstname')?.errors?.['required']">

Le prénom est obligatoire.

</small>

</div>

💡 Astuce UI : PrimeNG permet de styliser automatiquement les champs invalides avec pInputText et ng-invalid.

## Soumission du formulaire

ts

Copier

Modifier

submit(): void {

if (this.form.invalid) {

this.form.markAllAsTouched();

return;

}

const patient: Patient = this.form.value;

this.patientService.create(patient).subscribe({

next: () => this.router.navigate(['/patients']),

error: err => this.error = err.message

});

}

🔸 markAllAsTouched() force l’affichage des erreurs si l’utilisateur essaie de soumettre sans avoir rempli.

## Formulaires dynamiques et FormArray

Pour certaines entités complexes, le formulaire peut contenir un tableau dynamique de sous-champs, par exemple pour les traitements ou les antécédents.

ts

Copier

Modifier

treatments: this.fb.array([

this.fb.group({

name: ['', Validators.required],

dosage: ['']

})

])

👉 Cela permet d'ajouter ou retirer dynamiquement des groupes de champs.

## Liaison bidirectionnelle avec les modèles

Grâce aux types dans TypeScript, EyeWeb bénéficie d’un data-binding fortement typé entre :

le modèle Angular (Patient, Exam, SummaryDto…)

le FormGroup contenant les valeurs

le service qui envoie le FormGroup.value au backend

Cela réduit les erreurs de type et augmente la maintenabilité.

## Interactions conditionnelles dans le formulaire

Exemple d’activation conditionnelle d’un champ :

ts

Copier

Modifier

if (this.form.get('hasAllergy')?.value === true) {

this.form.get('allergyDetails')?.enable();

} else {

this.form.get('allergyDetails')?.disable();

}

🎯 Utilisé notamment dans les formulaires médicaux où certains champs sont contextuels.

## Résumé

EyeWeb utilise exclusivement des formulaires réactifs Angular pour toutes les interfaces complexes

Chaque formulaire est déclaré dans le fichier .ts via FormBuilder

Les validations sont riches, typées, et lisibles

Les erreurs sont capturées et affichées de manière explicite

Des composants visuels PrimeNG comme pInputText, pDropdown, pCalendar sont utilisés pour une UX fluide

Le backend reçoit des objets structurés, validés, et prêts à être persistés

# Routing, Navigation et Sécurisation des Routes dans Angular

La navigation est un des piliers fondamentaux des applications SPA (Single Page Application). Angular fournit un système de router intégré qui permet de configurer, sécuriser, paramétrer, et animer les navigations entre les différentes vues de l’application sans rechargement de la page. L’application EyeWeb exploite ce système en profondeur, avec une gestion modulaire, sécurisée et intuitive du routing client-side.

## Structure de base du routing

L'ensemble de la configuration de navigation repose sur le fichier app-routing.module.ts. Ce dernier déclare l'ensemble des chemins disponibles (path), les composants à afficher (component), et les restrictions d'accès éventuelles (canActivate, data, etc.).

ts

Copier

Modifier

const routes: Routes = [

{ path: '', component: HomeComponent },

{ path: 'login', component: LoginComponent },

{ path: 'summary/:id', component: SummaryComponent, canActivate: [AuthGuard] },

{ path: 'calculator', component: CalculatorComponent, canActivate: [AuthGuard] },

{ path: '\*\*', redirectTo: '' }

];

path: correspond à l’URL (ex: /summary/3)

component: composant Angular à afficher

canActivate: garde de sécurité (guard)

:id: paramètre de route dynamique

## Navigation dynamique dans le code

Angular permet la navigation à partir du code TypeScript grâce au service Router.

ts

Copier

Modifier

constructor(private router: Router) {}

goToSummary(patientId: number) {

this.router.navigate(['/summary', patientId]);

}

Ceci permet une navigation programmée (après une action utilisateur, une soumission, etc.), en opposition avec les liens statiques <a routerLink="/summary/3">.

## Paramètres de route

Lorsqu’un chemin inclut un :param, il peut être récupéré dans le composant cible via ActivatedRoute.

ts

Copier

Modifier

constructor(private route: ActivatedRoute) {}

ngOnInit(): void {

const id = this.route.snapshot.paramMap.get('id');

}

📌 EyeWeb utilise ceci pour charger dynamiquement un Patient, un Exam ou une SummaryDto depuis l’ID passé dans l’URL.

## Guards et sécurisation du routing

EyeWeb sécurise toutes les routes sensibles grâce à une garde d’authentification (guard) appelée AuthGuard, située dans le dossier guards.

ts

Copier

Modifier

canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean {

return this.authService.isAuthenticated();

}

Ce mécanisme permet :

de bloquer l’accès à certaines pages si l’utilisateur n’est pas connecté

de rediriger automatiquement vers /login si nécessaire

d’afficher dynamiquement ou non des éléments de navigation

## Redirections et routes par défaut

L’application est configurée pour rediriger vers la page d’accueil ou de login si l’URL n’est pas reconnue.

ts

Copier

Modifier

{ path: '\*\*', redirectTo: '', pathMatch: 'full' }

Ceci améliore l’UX et évite les erreurs 404 côté client.

## Lazy loading et modules enfants

Bien que non encore exploité en profondeur dans EyeWeb, Angular permet d’optimiser le routing avec du lazy loading (chargement paresseux des modules).

ts

Copier

Modifier

{ path: 'admin', loadChildren: () => import('./admin/admin.module').then(m => m.AdminModule) }

Cela permettrait, par exemple, de charger le module d’administration uniquement lorsque nécessaire, réduisant le poids initial de l'application.

## Synchronisation du routing et de l’état

Le Router est étroitement lié au service d’authentification de l’application. Lorsqu’un utilisateur se connecte ou se déconnecte, une redirection est immédiatement déclenchée :

ts

Copier

Modifier

this.authService.logout().subscribe(() => {

this.router.navigate(['/login']);

});

Cette cohérence garantit que l’état du client reste synchronisé avec les permissions du backend.

## Navigation conditionnelle dans l’interface

EyeWeb utilise les directives \*ngIf et les services d’authentification pour afficher ou non certains liens de navigation dans la barre de menu (navbar, sidenav, etc.).

html

Copier

Modifier

<li \*ngIf="authService.isAuthenticated()">

<a routerLink="/summary/3">Résumé</a>

</li>

## Avantages du système de routing Angular dans EyeWeb

Critère État dans EyeWeb

Navigation dynamique ✅ via Router

Paramètres de route ✅ (ActivatedRoute)

Sécurisation ✅ (canActivate)

Structure modulaire ✅ (Routes, AppRoutingModule)

Navigation UX fluide ✅ (pas de rechargement complet)

Affichage conditionnel ✅ selon rôle ou login

## Bonnes pratiques adoptées

Centralisation du routing dans un seul module

Préservation de l’authentification entre les redirections

Routes dynamiques avec :id

Redirection intelligente pour les chemins inconnus

Sécurité des composants sensibles avec AuthGuard

# Architecture des Services Angular : Communication avec l’API REST

Les services Angular représentent un élément fondamental dans l’architecture de toute application Angular professionnelle. Ils assurent la séparation des préoccupations (separation of concerns) en isolant la logique métier, les appels HTTP, la manipulation des données, et les règles de transformation. Dans l’application EyeWeb, cette séparation est appliquée rigoureusement et permet une communication fluide, réactive et typée avec les endpoints du backend Spring Boot.

## Structure générale des services

Chaque entité fonctionnelle de l'application (Patient, Calcul, Exam, Summary, Auth, etc.) possède un service dédié situé dans le dossier src/app/services/ ou dans des sous-dossiers liés à des modules fonctionnels.

Exemples :

auth.service.ts

patient.service.ts

summary.service.ts

calculator.service.ts

Tous les services suivent les bonnes pratiques Angular : décorés avec @Injectable({ providedIn: 'root' }), injectés dans les composants ou d’autres services via injection de dépendance, typés avec des interfaces ou classes modèles (ex : Patient, SummaryDto, Exam, etc.).

## Injection de HttpClient

L’ensemble des services utilise le module HttpClient fourni par Angular via @angular/common/http.

ts

Copier

Modifier

@Injectable({ providedIn: 'root' })

export class PatientService {

constructor(private http: HttpClient) {}

}

Ce service est capable d’effectuer toutes les opérations CRUD classiques sur les entités, mais aussi d’interagir avec des endpoints sécurisés, dynamiques, ou nécessitant un body JSON complexe.

## Exemple typique de service : Summary

Le fichier summary.service.ts permet de consommer le endpoint backend /api/summary/{id} en toute sécurité :

ts

Copier

Modifier

getSummaryByPatientId(id: number): Observable<SummaryDto> {

return this.http.get<SummaryDto>(`${environment.apiUrl}/summary/${id}`);

}

Utilisation de l’environnement (variable apiUrl) pour centraliser l’URL de base.

L’Observable<SummaryDto> garantit une réponse typée.

La méthode est réactive et peut être souscrite dans un composant (.subscribe() ou via async pipe).

## Typage strict et usage des modèles

Chaque service utilise des modèles typés définis dans des fichiers .ts spécifiques (par ex. summary.model.ts, patient.model.ts…), ce qui permet :

de bénéficier de l’intelliSense dans les IDE

de sécuriser les accès aux données

d’uniformiser la transformation des réponses HTTP

de détecter très tôt les erreurs de typage

## Observables et Réactivité

Tous les appels HTTP renvoient des Observables. Ce paradigme permet :

le chaînage d’opérateurs RxJS (map, tap, catchError, etc.)

la gestion asynchrone fluide (pas de callback hell)

le binding direct avec des templates via | async

Exemple :

ts

Copier

Modifier

this.summaryService.getSummaryByPatientId(3).subscribe(data => {

this.summary = data;

});

ou :

html

Copier

Modifier

<div \*ngIf="summaryService.getSummaryByPatientId(3) | async as summary">

<app-summary [data]="summary"></app-summary>

</div>

## Sécurité des requêtes : JWT dans HttpInterceptor

Les services n’ajoutent pas eux-mêmes le header Authorization: Bearer <token>. Cette tâche est centralisée dans un HttpInterceptor global qui intercepte toutes les requêtes sortantes :

ts

Copier

Modifier

intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

const token = this.authService.getToken();

if (token) {

const cloned = req.clone({

headers: req.headers.set('Authorization', 'Bearer ' + token)

});

return next.handle(cloned);

}

return next.handle(req);

}

## Résilience : gestion des erreurs HTTP

Chaque service peut intercepter les erreurs grâce à catchError, et éventuellement relancer la requête ou afficher une alerte.

ts

Copier

Modifier

this.http.get<Patient>(url).pipe(

catchError(error => {

console.error('Erreur API', error);

return throwError(() => new Error('Impossible de charger le patient.'));

})

);

## Cas particuliers de traitement : login et registration

Les services liés à l’authentification (dans auth.service.ts) gèrent des payloads particuliers :

ts

Copier

Modifier

authenticate(credentials: AuthRequest): Observable<AuthResponse> {

return this.http.post<AuthResponse>(`${apiUrl}/auth/authenticate`, credentials);

}

register(data: RegistrationRequest): Observable<AuthResponse> {

return this.http.post<AuthResponse>(`${apiUrl}/auth/register`, data);

}

Les tokens sont stockés dans localStorage, et récupérés automatiquement dans l’interceptor.

## Architecture REST First : respect des verbes HTTP

L’implémentation Angular respecte strictement les conventions REST :

Verbe HTTP Utilisation dans EyeWeb Exemple

GET Lire des données /summary/3, /patients

POST Créer une entité /auth/register, /patients

PUT Mettre à jour une entité existante /patients/5

DELETE Supprimer une entité /patients/5

Chaque verbe est respecté dans les services Angular, assurant une cohérence parfaite avec le backend Spring Boot.

## Environnement, modularité et centralisation

Le fichier environment.ts contient l’URL de base de l’API :

ts

Copier

Modifier

export const environment = {

production: false,

apiUrl: 'http://localhost:9292/api'

};

Toutes les routes y font référence : aucune URL n’est en dur dans les services. Cela permet une migration instantanée vers un environnement de production (déploiement sur serveur distant ou cloud).

## Résumé des avantages du système de services dans EyeWeb

Avantage Implémentation dans EyeWeb

Réutilisabilité Services injectables globalement (providedIn)

Réactivité Utilisation exclusive d’Observable

Sécurité Interceptor JWT intégré

Organisation Un service par entité logique

Typage strict Interfaces ou classes associées

Cohérence REST Respect des verbes HTTP

Gestion d’erreurs catchError, tap, retry, etc.

Découplage Composants = UI ; Services = données/API

# Architecture et fonctionnement des composants Angular

## Introduction au système de composants Angular

Dans Angular, tout repose sur les composants. Ce sont les briques fondamentales de l’interface utilisateur. Chaque composant :

dispose de sa propre logique TypeScript (.ts)

d’un template HTML (.html)

d’un style optionnel (.scss)

et interagit avec les services via l'injection de dépendances

EyeWeb exploite pleinement ce paradigme pour construire une interface médicale professionnelle, intuitive et dynamique.

## Organisation modulaire des composants

Les composants sont regroupés par fonctionnalité, en suivant le modèle Feature Module recommandé par Angular.

Exemples de dossiers fonctionnels :

/summary/ → Composants liés au récapitulatif opératoire

/calculator/ → Composants de calcul de lentilles

/patient/ → Composants de gestion des patients

/auth/ → Authentification (login, inscription)

Chaque module est associé à un fichier \*.module.ts (ex : summary.module.ts) qui importe et déclare ses composants localement, et exporte ceux nécessaires à l’extérieur.

## Schéma MVC côté client

Bien qu’Angular ne soit pas strictement un framework MVC, on peut résumer sa structure comme suit :

Élément Rôle dans EyeWeb

Component.ts Contrôleur + logique de gestion de données (MVVM)

Template.html Vue : l’interface visible, liée aux données via binding

Service.ts Modèle / couche d’accès aux données (API)

Cette approche garantit une séparation claire des responsabilités : un composant n’effectue aucun appel direct à la base, tout passe par les services.

## Composants réactifs avec RxJS

Tous les composants utilisent RxJS et le système d’observables pour gérer l’asynchronicité.

Exemple dans SummaryComponent :

ts

Copier

Modifier

summary$ = this.summaryService.getSummaryByPatientId(this.id);

Et dans le HTML :

html

Copier

Modifier

<app-summary-card \*ngIf="summary$ | async as summary" [data]="summary"></app-summary-card>

Avantages :

Pas besoin de subscribe() manuellement.

Composant automatiquement mis à jour à la réception de la réponse HTTP.

Syntaxe propre et testable.

## iaison de données (Data Binding)

EyeWeb utilise les quatre types de liaison Angular :

[property] : liaison unidirectionnelle (du composant vers la vue)

{{ expression }} : interpolation

(event) : écoute des événements (clics, entrées clavier…)

[(ngModel)] : liaison bidirectionnelle (dans les formulaires)

Exemple dans calculator.component.html :

html

Copier

Modifier

<input [(ngModel)]="lensPower" type="number" />

<button (click)="calculateIOL()">Calculer</button>

## Passage de données entre composants

L'application EyeWeb respecte le modèle hiérarchique Angular pour le partage de données :

Les composants parents transmettent les données aux enfants via @Input()

Les composants enfants informent les parents via @Output() + EventEmitter

Exemple dans summary.component.ts :

ts

Copier

Modifier

@Input() data: SummaryDto;

Et dans le parent :

html

Copier

Modifier

<app-summary [data]="summary$ | async"></app-summary>

## Affichage conditionnel et boucles

Les composants Angular s’appuient sur les directives structurelles :

\*ngIf : pour afficher une section si une condition est remplie

\*ngFor : pour générer dynamiquement des listes

ngSwitch : logique conditionnelle complexe

Exemple dans summary.component.html :

html

Copier

Modifier

<ul \*ngIf="data?.operatingMaterial?.length > 0">

<li \*ngFor="let item of data.operatingMaterial">{{ item }}</li>

</ul>

## Formulaires dynamiques et validation

L’application utilise à la fois :

les formulaires réactifs (FormBuilder, FormGroup) pour des cas avancés

les formulaires template-driven (ngModel) pour les cas simples

Chaque champ est validé côté client (avec contraintes HTML5, validators Angular) et côté serveur (réponses d’erreur).

## Design UI avec PrimeNG

La majorité des composants s’appuie sur PrimeNG, une bibliothèque de composants Angular professionnelle.

Composants utilisés :

p-table : pour afficher les listes de patients, d’examens, etc.

p-tree : pour gérer les antécédents et traitements (avec checkbox)

p-dialog : pour les formulaires en pop-up

p-toast : notifications

p-inputText, p-dropdown, p-calendar : formulaires

Avantages :

Design moderne

Comportements interactifs intégrés

Accessibilité

Responsive

## Routage Angular entre composants

EyeWeb utilise le système de router Angular pour naviguer entre les composants sans recharger la page :

Exemple de app-routing.module.ts :

ts

Copier

Modifier

const routes: Routes = [

{ path: 'summary/:id', component: SummaryComponent },

{ path: 'calculator', component: CalculatorComponent },

{ path: '\*\*', redirectTo: 'home' }

];

L’URL /summary/5 affiche automatiquement le résumé du patient 5.

Le paramètre id est capturé via ActivatedRoute :

ts

Copier

Modifier

this.id = this.route.snapshot.params['id'];

## Modularité & réutilisabilité

Chaque entité métier (Exam, Patient, Summary…) est encapsulée dans un composant dédié.

Les composants réutilisables sont extraits (ex : SummaryCardComponent) et peuvent être injectés n’importe où avec un simple selector.

## Détection des changements

Angular utilise la détection de changement automatique avec le moteur Zone.js. Les composants EyeWeb utilisent principalement la stratégie par défaut (ChangeDetectionStrategy.Default), ce qui suffit pour assurer la mise à jour de la vue à chaque mutation d’état.

## Résumé des bonnes pratiques appliquées

Pratique recommandée Mise en œuvre dans EyeWeb

Architecture modulaire Un composant par fonction, regroupés par module

Injection de services Tous les composants injectent les services via DI

Modèle asynchrone Utilisation exclusive d'observables (`

Design system PrimeNG + SCSS customisé

Séparation de la logique et de la vue TS = logique ; HTML = rendu

Réutilisation Composants réutilisables dans divers contextes

Routage paramétré ActivatedRoute pour capturer des paramètres

# Flux de données et gestion des états dans l’application Angular

## Objectif du flux de données

L’application EyeWeb repose sur un flux unidirectionnel de données, où :

le service agit comme source de vérité (données récupérées depuis l’API),

le composant consomme les données de manière asynchrone,

la vue s’adapte automatiquement via l’utilisation de l’opérateur | async.

Ce modèle favorise une architecture prédictible, testable et maintenable, essentielle dans une application médicale.

## Principes de gestion d’état Angular

Angular ne fournit pas de système de gestion d’état global natif (comme Redux), mais exploite RxJS pour gérer des états locaux (par composant) ou partagés (via services).

EyeWeb suit ces principes :

Pas de variables mutables dans les composants.

Les services utilisent des BehaviorSubject ou Observable comme unique source de données.

Le composant souscrit avec | async pour garder la vue synchronisée.

## Exemple de flux complet dans EyeWeb

* Étape 1 : Action utilisateur

Le médecin clique sur un patient → déclenche un router.navigate() vers /summary/:id.

* Étape 2 : Chargement dans SummaryComponent

ts

Copier

Modifier

summary$ = this.summaryService.getSummaryByPatientId(this.id);

Ici, summary$ est un Observable<SummaryDto> exposé par le service, qui encapsule la requête HTTP.

* Étape 3 : Appel du service

ts

Copier

Modifier

getSummaryByPatientId(id: number): Observable<SummaryDto> {

return this.http.get<SummaryDto>(`${this.apiUrl}/summary/${id}`);

}

Le service joue le rôle d’intermédiaire entre le backend et le frontend.

* Étape 4 : Affichage dans la vue HTML

html

Copier

Modifier

<app-summary-card \*ngIf="summary$ | async as summary" [data]="summary"></app-summary-card>

L'opérateur | async souscrit automatiquement à l'observable et réagit aux changements de flux.

## Utilisation des Observable et Subject

Cas simple : Observable statique

ts

Copier

Modifier

getAllPatients(): Observable<Patient[]> {

return this.http.get<Patient[]>(this.baseUrl + '/patients');

}

Cas avancé : stockage d’état avec BehaviorSubject

ts

Copier

Modifier

private selectedPatientSubject = new BehaviorSubject<Patient | null>(null);

selectedPatient$ = this.selectedPatientSubject.asObservable();

selectPatient(patient: Patient) {

this.selectedPatientSubject.next(patient);

}

Ce pattern est utilisé pour :

mémoriser l’état d’un patient sélectionné

le partager entre plusieurs composants

l’utiliser dans un autre module sans le recharger

## Synchronisation composant-service

Les composants n'ont aucune logique métier :

Ils se contentent d'écouter les données exposées par les services (observable$)

Ils appellent des méthodes d’action (selectPatient(), calculateIOL()...) via des événements ((click))

Cela permet une isolation stricte entre l’affichage et le traitement.

## Système de notifications (flux à part)

Les services utilisent aussi RxJS pour propager des événements comme les messages de confirmation :

ts

Copier

Modifier

private toastSubject = new Subject<string>();

toast$ = this.toastSubject.asObservable();

showMessage(message: string) {

this.toastSubject.next(message);

}

Et dans le ToastComponent :

ts

Copier

Modifier

this.service.toast$.subscribe(msg => this.message = msg);

## État local vs global

Type d’état Localisation dans EyeWeb Portée Exemple

État local Variable de composant (selectedTab) Composant Onglet actif, champ temporaire

État partagé Service injectable (RxJS Subject) Application entière Patient sélectionné

État distant Backend (via API REST) Global, persistant Résumé opératoire

## Usage avancé : switchMap, combineLatest, map

Certains composants combinent plusieurs flux :

ts

Copier

Modifier

this.data$ = this.route.params.pipe(

switchMap(params => this.service.getSummaryByPatientId(+params['id']))

);

Ici :

params est observable (route dynamique)

switchMap permet de remplacer l’appel HTTP précédent si l’utilisateur change d’ID rapidement

D'autres cas utilisent combineLatest pour synchroniser plusieurs sources (ex : filtre + liste active).

## Recommandations appliquées dans EyeWeb

Bonne pratique Mise en œuvre

Un flux = une responsabilité Un observable = un type de données

Éviter les abonnements manuels (subscribe()) Utilisation de `

État centralisé dans les services Utilisation de BehaviorSubject

Pas de mutation directe Données transmises via .next()

Réactivité sur les changements dynamiques switchMap, combineLatest

## Conclusion du flux RxJS dans EyeWeb

L’architecture de données de l’application repose sur un paradigme réactif complet, bien intégré avec le système Angular :

Le backend expose une API REST sécurisée

Le service frontend interagit avec cette API

Le composant observe les données et déclenche les actions

La vue s’adapte automatiquement sans mutabilité

Cette approche garantit des composants légers, une architecture prévisible, un code testable, et une grande évolutivité.

# Système de Formulaires Angular dans EyeWeb

## Rôle central des formulaires

Dans l’application EyeWeb, les formulaires sont essentiels pour :

Créer et modifier des entités médicales (Patients, Exams, Calculs, etc.).

Gérer l’authentification (login, inscription, changement de mot de passe).

Saisir des données médicales précises et validées (antécédents, lentilles, chirurgies, etc.).

Angular propose deux approches :

Formulaires Template-driven (déclaratif, moins utilisé ici).

Formulaires Reactive (Reactive Forms) : plus puissants, utilisés dans EyeWeb.

## Déclaration du FormGroup

Les formulaires réactifs sont construits dans le .ts du composant via FormBuilder :

ts

Copier

Modifier

this.examForm = this.fb.group({

examDate: [null, Validators.required],

eyeSide: [null, Validators.required],

al: [null, [Validators.required, Validators.min(10), Validators.max(40)]],

lensThickness: [null],

k1: [null],

k2: [null]

});

🔍 Chaque champ est contrôlé par un FormControl, associé à une valeur par défaut et des validateurs.

## Affichage dans le template HTML

Le FormGroup est relié au template via [formGroup] :

html

Copier

Modifier

<form [formGroup]="examForm" (ngSubmit)="onSubmit()">

<input type="date" formControlName="examDate" />

<select formControlName="eyeSide">

<option value="OD">OD</option>

<option value="OS">OS</option>

</select>

<input type="number" formControlName="al" />

<button type="submit" [disabled]="examForm.invalid">Valider</button>

</form>

✅ L'état du formulaire est synchronisé automatiquement grâce au binding réactif.

## Contrôles et validation

EyeWeb applique des validations standards :

Validators.required : champ obligatoire

Validators.min / max : valeur numérique dans un intervalle

Validators.email : format d’email

Custom validators : possibles pour les valeurs médicales complexes (ex : ACD > 0, K1 < K2, etc.)

Exemple d’affichage conditionnel des erreurs :

html

Copier

Modifier

<input formControlName="al" type="number" />

<div \*ngIf="examForm.get('al').invalid && examForm.get('al').touched">

Longueur axiale invalide

</div>

## Soumission du formulaire

Lors du clic sur "Valider", le code exécute :

ts

Copier

Modifier

onSubmit() {

if (this.examForm.valid) {

const examData = this.examForm.value;

this.examService.createExam(examData).subscribe(...);

}

}

📤 Le contenu du formulaire est directement utilisable en tant qu’objet JSON.

## Mise à jour (patching) d’un formulaire

Lors d’un edit :

ts

Copier

Modifier

this.examService.getExam(id).subscribe(exam => {

this.examForm.patchValue({

examDate: exam.examDate,

eyeSide: exam.eyeSide,

al: exam.al

});

});

🧠 patchValue() permet de pré-remplir le formulaire avec les données existantes.

## Dynamisme dans les champs

EyeWeb utilise des champs dynamiques en fonction :

De la valeur d’un autre champ (ex : œil dominant, antécédents)

De la sélection du médecin

Exemple :

ts

Copier

Modifier

this.examForm.get('eyeSide')?.valueChanges.subscribe(value => {

if (value === 'OD') {

this.examForm.get('dominantEye')?.setValue(true);

}

});

## Intégration avec PrimeNG

EyeWeb combine Reactive Forms avec PrimeNG pour un rendu ergonomique :

html

Copier

Modifier

<p-inputNumber formControlName="k1" [min]="30" [max]="60"></p-inputNumber>

<p-dropdown formControlName="machine" [options]="biometers"></p-dropdown>

<p-calendar formControlName="examDate"></p-calendar>

🎨 PrimeNG applique des styles, de la validation visuelle, et améliore l’expérience utilisateur.

## Messages de validation et feedback utilisateur

Chaque soumission correcte ou erronée génère une notification (toast) :

ts

Copier

Modifier

this.messageService.add({ severity: 'success', summary: 'Formulaire validé' });

Le service est injecté depuis PrimeNG (MessageService) et déclenché dans la méthode onSubmit.

## Bonnes pratiques respectées dans EyeWeb

Bonne pratique Mise en œuvre dans EyeWeb

Tous les champs dans un FormGroup ✅ examForm, userForm, etc.

Validateurs définis côté TS, erreurs visibles côté HTML ✅ avec formControlName et \*ngIf

Patching des formulaires pour l’édition ✅ via patchValue()

Envoi conditionné par form.valid ✅ if (form.valid)

Affichage dynamique, conditionnel ✅ avec valueChanges()

Couplage avec PrimeNG ✅ pour tous les inputs, select, calendar

Réutilisabilité via composants ✅ SummaryFormComponent, PatientFormComponent, etc.

# Module Summary (Vue opératoire du patient)

## Objectif et rôle du module

Le composant Summary constitue un hub central d'informations dans EyeWeb. Il synthétise toutes les données médicales pertinentes du patient sélectionné, notamment :

Informations administratives du patient.

Résultats de l’examen ophtalmologique.

Choix de la lentille (IOL).

Données sur la chirurgie prévue.

Traitements avant/après opération.

Matériel utilisé.

Il s'agit d’un composant read-only (lecture seule), affichant une vue unifiée et consolidée des différentes entités liées à un patient donné.

## Architecture du module

Le module summary est un sous-module Angular dédié, avec la structure suivante :

css

Copier

Modifier

src/

└── app/

└── summary/

├── summary.component.ts

├── summary.component.html

├── summary.component.scss

├── summary.service.ts

├── summary-routing.module.ts

└── summary.module.ts

Ce découpage assure :

Une séparation claire avec les autres modules.

Un chargement potentiel en lazy-loading.

Une maintenabilité optimale.

## Service summary.service.ts

Ce service Angular appelle l’API backend (/api/summary/{patientId}) et transforme la réponse JSON en SummaryDto.

ts

Copier

Modifier

getSummaryByPatientId(id: number): Observable<SummaryDto> {

return this.http.get<SummaryDto>(`${this.apiUrl}/summary/${id}`);

}

✅ Ce service est injecté dans SummaryComponent et isolé de la logique de présentation.

## Composant summary.component.ts

Dans le fichier TypeScript du composant, la logique principale est :

ts

Copier

Modifier

ngOnInit() {

this.route.params.subscribe(params => {

const patientId = +params['id'];

this.summaryService.getSummaryByPatientId(patientId).subscribe(summary => {

this.summary = summary;

});

});

}

Le SummaryDto est donc injecté dynamiquement au chargement de la page en fonction de l’ID du patient dans l’URL.

## Fichier HTML (summary.component.html)

Le template regroupe toutes les sections du résumé opératoire, découpées en panneaux :

html

Copier

Modifier

<p-panel header="Informations Patient">

<div>Nom : {{ summary.name }}</div>

<div>Prénom : {{ summary.firstName }}</div>

<div>Date de naissance : {{ summary.dob }}</div>

<div>Profession : {{ summary.profession }}</div>

</p-panel>

<p-panel header="Examen Ophtalmologique">

<div>Œil concerné : {{ summary.laterality }}</div>

<div>Œil dominant : {{ summary.dominantEye }}</div>

<div>Allergies : {{ summary.allergy }}</div>

<div>Antécédents : {{ summary.atcdO }}</div>

<div>Histoire médicale : {{ summary.history }}</div>

</p-panel>

<p-panel header="Lentille et Calcul">

<div>Lentille sélectionnée : {{ summary.selectedIOL }}</div>

<div>Puissance : {{ summary.power }}</div>

<div>Fabricant : {{ summary.manufacturer }}</div>

</p-panel>

<p-panel header="Chirurgie">

<div>Type : {{ summary.surgeryType }}</div>

<div>Chirurgien : {{ summary.surgeon }}</div>

<div>Date : {{ summary.surgeryDate }}</div>

<div>Anesthésie : {{ summary.anesthesia }}</div>

<div>Urgence : {{ summary.emergency }}</div>

<div>Rappel : {{ summary.recall }}</div>

</p-panel>

🌐 Les données sont donc entièrement dynamiques et proviennent du backend.

## Intégration avec le routing

Dans le summary-routing.module.ts :

ts

Copier

Modifier

const routes: Routes = [

{ path: ':id', component: SummaryComponent }

];

📍 Le composant est activé via l’URL /summary/:id, et utilise le router param id pour interroger le backend.

## Couplage avec PrimeNG

Le composant summary utilise intensément :

p-panel pour l'organisation visuelle.

p-divider, p-tag, p-tooltip (optionnel) pour plus d’ergonomie.

p-card ou p-scrollPanel pour une structure responsive.

Cette UI claire facilite la compréhension des données par les ophtalmologues et le personnel médical.

## Avantages de cette approche

Avantage Détail

Centralisation Tous les aspects cliniques d’un patient sont rassemblés.

Séparation des responsabilités Le backend agrège les données, le frontend se contente de les afficher.

Réactivité La vue s’adapte automatiquement à l’ID dans l’URL.

Maintenance Le composant est isolé et peut évoluer indépendamment.

Esthétique UI moderne grâce à PrimeNG.

## Lien avec le backend

Le composant repose entièrement sur la méthode Java getSummaryByPatientId(Long id) dans CataractServiceImpl, qui compose un SummaryDto à partir de :

Patient

Exam

Calcul

Lens

✅ La structure du DTO est plate (pas d’objet imbriqué), ce qui facilite le mapping côté Angular.

## Perspectives d'évolution

Ajout d’un bouton "Exporter en PDF" du résumé.

Intégration directe dans la fiche du patient (PatientDetailComponent).

Affichage différencié OD/OS si double chirurgie.

Ajout d'une étape de signature électronique par le médecin.

# Module Patient (Gestion des patients)

## Objectif fonctionnel

Le module Patient permet la gestion complète des utilisateurs suivis par l’application. Il centralise toutes les informations administratives et médicales de base permettant :

L'identification du patient.

La consultation de son historique médical.

La liaison avec ses examens (Exam) et calculs (Calcul).

L'accès à ses données opératoires (via Summary).

## Structure modulaire (Angular)

Le module est organisé comme suit dans l’arborescence Angular :

css

Copier

Modifier

src/

└── app/

└── patient/

├── patient.component.ts

├── patient.component.html

├── patient.service.ts

├── patient-routing.module.ts

├── patient.module.ts

├── patient.model.ts

├── patient-form/

│ ├── patient-form.component.ts

│ └── patient-form.component.html

└── patient-list/

├── patient-list.component.ts

└── patient-list.component.html

📦 Cette architecture modulaire respecte la séparation des responsabilités :

patient-list : composant de listing dynamique.

patient-form : composant de création / édition.

patient.service : service de communication HTTP.

patient.model.ts : modèle de données TypeScript.

## Fonctionnalités couvertes

Fonction Description

Création Formulaire réactif avec validations.

Lecture Tableau dynamique avec tri, filtre et pagination.

Mise à jour Edition dans un formulaire pré-rempli.

Suppression Suppression logique (soft-delete ou suppression physique selon la config).

Navigation Redirection vers les modules liés (Exam, Summary, etc.).

## Intégration des composants PrimeNG

Le module utilise les composants suivants de PrimeNG :

p-table pour l’affichage de la liste de patients.

p-dialog pour les formulaires en modale.

p-inputText, p-calendar, p-dropdown pour les champs du formulaire.

p-toast pour les messages de succès/erreur.

Exemple :

html

Copier

Modifier

<p-table [value]="patients" [paginator]="true" [rows]="10">

<ng-template pTemplate="header">

<tr>

<th>Nom</th>

<th>Prénom</th>

<th>Date de naissance</th>

<th>Email</th>

<th>Actions</th>

</tr>

</ng-template>

<ng-template pTemplate="body" let-patient>

<tr>

<td>{{ patient.lastname }}</td>

<td>{{ patient.firstname }}</td>

<td>{{ patient.birthDate | date }}</td>

<td>{{ patient.email }}</td>

<td>

<button (click)="edit(patient)">Modifier</button>

<button (click)="delete(patient.id)">Supprimer</button>

</td>

</tr>

</ng-template>

</p-table>

## Formulaire de patient (patient-form.component)

Le formulaire utilise ReactiveFormsModule avec les champs suivants

firstname

lastname

email

birthDate

phoneNumber

job

language

gender

Avec des validations Angular intégrées (Validators.required, Validators.email, etc.) et feedback visuel via PrimeNG.

## Service Angular PatientService

Service injectable centralisant tous les appels HTTP à l’API backend :

ts

Copier

Modifier

getAll(): Observable<Patient[]> {

return this.http.get<Patient[]>(`${this.apiUrl}/patients`);

}

create(patient: Patient): Observable<Patient> {

return this.http.post<Patient>(`${this.apiUrl}/patients`, patient);

}

update(patient: Patient): Observable<Patient> {

return this.http.put<Patient>(`${this.apiUrl}/patients/${patient.id}`, patient);

}

delete(id: number): Observable<void> {

return this.http.delete<void>(`${this.apiUrl}/patients/${id}`);

}

27.7 Intégration backend

Le contrôleur PatientController dans Spring Boot expose les endpoints REST suivants :

GET /api/patients : liste complète.

GET /api/patients/{id} : un patient.

POST /api/patients : création.

PUT /api/patients/{id} : mise à jour.

DELETE /api/patients/{id} : suppression.

Ce contrôleur utilise le service PatientService, qui interagit avec le repository JPA pour exécuter les requêtes sur la table patient.

## Routage Angular

Le patient-routing.module.ts configure les routes suivantes :

ts

Copier

Modifier

const routes: Routes = [

{ path: '', component: PatientListComponent },

{ path: 'new', component: PatientFormComponent },

{ path: ':id/edit', component: PatientFormComponent }

];

L’utilisateur peut :

Voir la liste des patients à /patients.

Créer un patient via /patients/new.

Modifier un patient via /patients/:id/edit.

## Connexion avec les autres modules

Le module Patient joue un rôle pivot :

Le Patient.id est utilisé comme clé étrangère dans Exam, Calcul, Lens, etc.

La fiche patient permet l’accès aux modules Summary, Exam, Calculator.

➡️ Ce lien bidirectionnel assure une navigation fluide et cohérente dans l’application.

## Évolutions futures possibles

Amélioration Description

Recherche intelligente Intégrer une recherche dynamique par nom, prénom ou email.

Pagination serveur Adapter le back pour une meilleure scalabilité.

Filtres multiples Ajout de filtres par langue, âge, genre.

Fusion patients En cas de doublon, permettre une fusion semi-automatisée.

# Module Calculator (Calculs biométriques et sélection de l’IOL)

## Objectif fonctionnel

Le module Calculator gère les calculs biométriques destinés à déterminer la lentille intraoculaire (IOL) optimale à implanter chez un patient atteint de cataracte.

Ce module est au cœur du processus opératoire. Il prend en compte plusieurs paramètres mesurés lors de l'examen (kératométrie, ACD, AL, etc.) ainsi que des données cliniques pour suggérer une ou plusieurs lentilles adaptées.

## Intégration dans le workflow patient

Ce module est accessible après la sélection d’un Exam lié à un Patient. Il est déclenché soit automatiquement lors de l’enregistrement d’un nouvel examen, soit manuellement via l’interface Angular :

ts

Copier

Modifier

this.router.navigate(['/calculator', exam.id]);

Il s’agit donc d’un module dépendant de l’ID d’un Exam et interagit avec les entités Lens et Calcul.

## Structure modulaire Angular

Le module Angular est structuré ainsi :

cpp

Copier

Modifier

src/

└── app/

└── calculator/

├── calculator.component.ts

├── calculator.component.html

├── calculator.service.ts

├── calculator.model.ts

├── calculator.module.ts

Chaque composant a une responsabilité claire :

calculator.component : affichage des résultats et formulaire de saisie.

calculator.service : communication avec l’API backend.

calculator.model.ts : définition des interfaces Calcul, Lens, etc.

## Fonctionnalités métier

Fonction Description

Sélection automatique de l’IOL Suggestion de lentille basée sur les constantes biométriques.

Calculs multiples Génération de plusieurs scénarios via des formules différentes (Haigis, SRK/T, etc.).

Validation L'utilisateur peut sélectionner une lentille parmi celles proposées.

Mise à jour du calcul Possibilité de modifier certains paramètres manuellement.

Association au dossier Le calcul sélectionné est sauvegardé et lié à l’examen.

## Intégration backend (Spring Boot)

Entité Calcul

Liée à Exam par une relation @ManyToOne.

Contient les paramètres : formule, puissance, type de lentille (Lens), date, sélection, etc.

Un exam peut avoir plusieurs Calcul, mais un seul peut être selected = true.

Entité Lens

Table de référence des lentilles disponibles.

Chaque lentille a une power, un name, une manufacturer, un aConstant, un lensType.

Repository & Services

java

Copier

Modifier

List<Calcul> findByExamId(Long examId);

Optional<Calcul> findByExamIdAndSelectedTrue(Long examId);

La sélection est faite dans le service CalculatorServiceImpl, avec une logique métier personnalisée pour attribuer la bonne lentille.

## Algorithme de sélection de lentille

L'algorithme s’appuie sur :

La kératométrie (k1, k2)

La profondeur de chambre antérieure (acd)

La longueur axiale (al)

La formule choisie (ex: SRK-T, Haigis, Holladay 2…)

Exemple de logique (simplifiée) en backend :

java

Copier

Modifier

public Calcul calculerSRKT(Exam exam, Lens lens) {

double A = lens.getAConstant();

double AL = exam.getAl();

double K = (exam.getK1() + exam.getK2()) / 2;

double power = A - 2.5 \* AL - 0.9 \* K;

return new Calcul(exam, lens, power);

}

🧠 Des constantes biométriques et des formules avancées peuvent être intégrées selon les besoins cliniques réels.

## Interface Angular

L’interface utilisateur comprend :

Une liste des calculs précédents.

Un tableau de suggestions (PrimeNG p-table).

Une vue des détails du calcul sélectionné.

Un bouton pour valider le choix de l’IOL.

Exemple de binding :

html

Copier

Modifier

<p-table [value]="calculs">

<ng-template pTemplate="body" let-c>

<tr [class.selected]="c.selected">

<td>{{ c.formula }}</td>

<td>{{ c.lens.name }}</td>

<td>{{ c.power | number:'1.2-2' }} D</td>

<td><button (click)="select(c)">Sélectionner</button></td>

</tr>

</ng-template>

</p-table>

28.8 Service Angular CalculatorService

Fonctions essentielles :

getCalculsByExamId(examId: number): Observable<Calcul[]>

selectCalcul(calculId: number): Observable<void>

createCalcul(calcul: Calcul): Observable<Calcul>

Chaque appel repose sur les endpoints exposés par le backend via REST.

## Navigation & Sécurité

L’accès au module est restreint par rôle (ROLE\_OPHTALMO).

Les routes sont protégées par guards Angular (AuthGuard).

Le token JWT est attaché à chaque requête.

## Piste d’amélioration future

Idée Description

Formules personnalisées Intégration d’un système permettant au chirurgien de définir ses propres formules.

Export PDF Génération d’un document résumant les calculs effectués.

Mode simulation Comparaison des lentilles sur plusieurs paramètres visuels (Myopie, Hyperopie...).

# Module Summary (Résumé opératoire du patient)

## Objectif fonctionnel

Le module Summary permet de centraliser toutes les données cliniques nécessaires à la planification de l’intervention chirurgicale. Il s’agit d’un point d’entrée essentiel pour le chirurgien, lui offrant une vision synthétique et fiable du patient, de son état ophtalmologique, et des décisions prises en amont (calcul de la lentille, traitements, antécédents...).

Il fonctionne comme une fiche opératoire digitale, à usage médical.

## Étapes d’accès

L’utilisateur accède au module Summary à partir de la liste des patients, puis en sélectionnant un patient ayant au moins un Exam associé. L’ID du patient est transmis au composant Summary via le routeur Angular :

ts

Copier

Modifier

this.router.navigate(['/summary', patient.id]);

## Structure Angular du module

Le module est structuré en Angular selon cette arborescence :

css

Copier

Modifier

src/

└── app/

└── summary/

├── summary.component.ts

├── summary.component.html

├── summary.model.ts

├── summary.service.ts

└── summary.module.ts

Chaque fichier assure une responsabilité unique :

component.ts : logique de récupération et de binding des données.

component.html : mise en page visuelle avec PrimeNG.

service.ts : communication HTTP avec le backend.

model.ts : définition stricte du DTO Summary.

module.ts : déclaration des dépendances Angular.

## Backend — Génération du Summary

La méthode principale est :

java

Copier

Modifier

SummaryDto getSummaryByPatientId(Long patientId)

Elle est définie dans le service CataractServiceImpl et agrège des données issues de :

Source Données utilisées

Patient Nom, prénom, profession, date de naissance

Exam Antécédents, traitements, œil dominant, œil opéré, date opération, type chirurgie

Calcul Lentille sélectionnée, fabricant, puissance

Lens Paramètres optiques

Autres champs Valeurs mockées en attendant les entités (matériel, traitements post-op...)

L’examen sélectionné est récupéré avec :

java

Copier

Modifier

Optional<Exam> selectedExam = cataractRepository.findSelectedExamByPatientId(patientId);

Et le calcul sélectionné avec :

java

Copier

Modifier

Calcul calcul = selectedExam.getCalculs().stream()

.filter(Calcul::isSelected)

.findFirst()

.orElse(null);

## DTO et sérialisation

Le SummaryDto centralise toutes les informations et permet une transmission propre et typée entre le backend et le frontend :

java

Copier

Modifier

public class SummaryDto {

private String name;

private String firstName;

private String dob;

private String profession;

private String atcdO;

private String history;

private String allergy;

private String dominantEye;

private String surgeryType;

private String surgeon;

private String laterality;

private String surgeryDate;

private String recall;

private String emergency;

private String anesthesia;

private String selectedIOL;

private String manufacturer;

private String power;

private String[] operatingMaterial;

private String[] preoperativeTreatment;

private String[] postoperativeTreatment;

private String[] postoperativeAppointments;

}

## Interface utilisateur Angular

Le composant summary.component.html affiche les données dans une interface utilisateur propre et accessible, en utilisant :

PrimeNG Card pour la mise en page générale.

GridLayout (p-grid) pour l’alignement des informations.

p-badge pour signaler les informations urgentes.

p-timeline ou p-accordion pour organiser les informations chirurgicales.

Exemple d’extrait HTML :

html

Copier

Modifier

<p-card header="Résumé opératoire">

<div class="p-grid">

<div class="p-col-4"><b>Nom :</b> {{ summary.name }}</div>

<div class="p-col-4"><b>Prénom :</b> {{ summary.firstName }}</div>

<div class="p-col-4"><b>Date de naissance :</b> {{ summary.dob }}</div>

</div>

...

</p-card>

## Service Angular

Le SummaryService permet de récupérer le résumé via un appel sécurisé :

ts

Copier

Modifier

getSummary(patientId: number): Observable<SummaryDto> {

return this.http.get<SummaryDto>(`${this.apiUrl}/summary/${patientId}`);

}

Le token JWT est automatiquement ajouté à l’appel par l’intercepteur de sécurité.

## Sécurité et rôles

L'accès au résumé est sécurisé :

Le backend vérifie les rôles (ROLE\_OPHTALMO, ROLE\_ADMIN).

Le frontend restreint l’accès aux utilisateurs authentifiés (Guard).

Les données sensibles sont filtrées selon le rôle, si nécessaire.

## Problématiques techniques rencontrées

Problème Solution

Champs manquants (matériel, traitements) Valeurs mockées ou arrays vides

Erreurs de sérialisation Correction via Jackson (@JsonProperty)

Multiples Exam sans sélection Ajout de la notion selected = true

Surcharge du DTO Ajout progressif des champs avec rétrocompatibilité

29.10 Améliorations prévues

Amélioration Objectif

Liaison aux vrais traitements post-opératoires Intégration d’une entité Treatment

Refonte graphique Timeline opératoire, export PDF

Ajout du chirurgien connecté Lier la session à l’utilisateur opérant

Synthèse vocale ou résumé imprimable Accessibilité patient / personnel médical

# Gestion des utilisateurs et des rôles

## Objectif

Le module de gestion des utilisateurs assure l’ensemble des responsabilités suivantes :

Authentification sécurisée (connexion, inscription, mot de passe oublié).

Attribution dynamique de rôles (admin, médecin, secrétaire...).

Protection des endpoints sensibles via des filtres et annotations.

Stockage et traçabilité des tokens JWT.

Gestion du cycle de vie des comptes (création, activation, révocation, suppression...).

Il s’agit du cœur de la sécurité applicative, garantissant que seuls les utilisateurs autorisés accèdent aux fonctionnalités cliniques.

## Entité User (utilisateur)

Chaque utilisateur est représenté par une entité persistée, enrichie d’attributs utiles à l’authentification, à l’autorisation, à la personnalisation et à la gestion de session.

Champs principaux :

Attribut Description

id Identifiant technique

email Identifiant unique (login)

password Mot de passe hashé (BCrypt)

role Enum Role (ADMIN, OPHTALMO, SECRETAIRE)

isEnabled Compte activé ou désactivé

verificationCode Code d’activation à usage unique

tokens Liste de tokens JWT actifs pour cet utilisateur

L’utilisateur est sérialisé via Jackson et sécurisé par UserDetailsServiceImpl.

## Enum Role

L’énumération des rôles permet de déterminer l’accès aux différentes parties de l’application :

java

Copier

Modifier

public enum Role {

ADMIN,

OPHTALMO,

SECRETAIRE

}

Chaque rôle dispose de ses propres privilèges définis dans les fichiers de configuration Spring Security (classe SecurityConfig).

30.4 Authentification et service utilisateur

L’implémentation repose sur un ensemble de composants coordonnés :

Classe / Composant Rôle

AuthenticationServiceImpl Logique métier : login, register, logout

JwtServiceImpl Génération et validation des tokens

LogoutServiceImpl Révocation des tokens

UserDetailsServiceImpl Chargement d’un utilisateur par son email

TokenRepository Accès aux tokens stockés (revocation, historique)

AuthenticationController Contrôleur REST exposant les endpoints /auth/\*\*

30.5 Sécurisation des endpoints

Les routes sont protégées dans SecurityConfig via une configuration claire :

java

Copier

Modifier

http

.authorizeHttpRequests(auth -> auth

.requestMatchers("/auth/\*\*").permitAll()

.requestMatchers("/admin/\*\*").hasRole("ADMIN")

.requestMatchers("/exam/\*\*").hasAnyRole("OPHTALMO", "ADMIN")

.anyRequest().authenticated()

)

Cette hiérarchisation garantit que :

L’inscription/connexion est publique.

La gestion des données médicales est réservée aux rôles cliniques.

Les tâches d’administration (droits, utilisateurs) sont strictement réservées à l’ADMIN.

## Cycle d’authentification complet

Voici un résumé du cycle utilisateur, de son inscription à la révocation de session :

### Inscription (/auth/register)

L’utilisateur soumet son email, mot de passe, rôle.

Le mot de passe est hashé avec BCrypt.

Un verificationCode est généré pour validation par email.

Un JWT est retourné dès inscription, stocké dans la table Token.

### Connexion (/auth/authenticate)

L’utilisateur soumet ses credentials.

L’authentification est vérifiée via AuthenticationManager.

Le JWT est généré et envoyé.

Le token est persisté en base (valide, non expiré).

### Utilisation des ressources

Le frontend envoie le JWT dans l’en-tête Authorization: Bearer <token>.

Le JwtAuthenticationFilter intercepte et valide ce token.

L’utilisateur est injecté dans le SecurityContext.

### Déconnexion (/auth/logout)

Appel au LogoutServiceImpl

Révocation logique du token courant (marqué comme expiré ou révoqué).

## Table Token et suivi des connexions

La table Token en base permet de stocker les JWT en circulation, assurant une meilleure traçabilité et une possibilité de révocation centralisée.

Champs de Token :

Champ Description

token JWT encodé

revoked Booléen indiquant une révocation explicite

expired Booléen si le token a dépassé sa date de validité

user Relation vers l’entité User

Un token est considéré valide si :

Il n’est pas expiré.

Il n’a pas été révoqué.

Il est lié à un utilisateur actif.

## Gestion frontend des rôles et sessions

Dans l’interface Angular :

Le JWT est stocké en sessionStorage ou localStorage.

Un interceptor HTTP ajoute le token à chaque requête sortante.

Un guard Angular bloque l’accès aux routes si l’utilisateur n’est pas authentifié.

Les composants affichent des éléments conditionnellement au rôle (\*ngIf="isAdmin()").

## Vue d’administration des utilisateurs (TODO)

Une interface d’administration est envisagée pour :

Lister les utilisateurs avec rôle.

Modifier les statuts (actif / désactivé).

Réinitialiser des mots de passe.

Révoquer manuellement des tokens actifs.

# Calcul de la lentille intraoculaire (IOL)

## Objectif

Dans le cadre de la chirurgie de la cataracte, le choix d’une lentille intraoculaire (IOL – Intra Ocular Lens) est essentiel pour garantir une vision optimale post-opératoire. EyeWeb permet :

La saisie et l’enregistrement des calculs optiques associés à un examen donné.

L’association entre un calcul et une lentille spécifique.

L’identification du calcul sélectionné pour l’intervention.

La traçabilité des formules, puissances, constantes et paramètres biométriques associés.

## Structure de l’entité Calcul

L’entité Calcul représente un calcul IOL spécifique lié à un examen donné. Elle contient :

Attribut Description

id Identifiant technique

exam L’examen ophtalmologique auquel le calcul est rattaché

lens La lentille sélectionnée dans ce calcul

formulaUsed Formule de calcul utilisée (ex : SRK/T, Hoffer Q, Barrett...)

powerCalculated Puissance optique calculée

selected Booléen indiquant si ce calcul est celui retenu pour la chirurgie

Remarque : Plusieurs calculs peuvent exister pour un même examen, mais un seul est marqué comme selected = true.

## Structure de l’entité Lens

La classe Lens contient les détails techniques de la lentille intraoculaire :

Attribut Description

id Identifiant

name Nom commercial de la lentille

manufacturer Fabricant

power Puissance (en dioptries)

material Matériau

toric Torique ou non

addPower Add pour les multifocales (si applicable)

constantA Constante A utilisée dans la formule de calcul

La relation avec Calcul est de type @ManyToOne, ce qui signifie qu’une même lentille peut être réutilisée dans plusieurs calculs.

## Fonctionnalités métier

Lorsqu’un examen biométrique est terminé, l’ophtalmologue ou son assistant peut créer plusieurs scénarios de calculs.

Chaque scénario peut tester une formule différente, une IOL différente, ou des objectifs réfractifs personnalisés.

Un calcul unique est sélectionné via le champ selected, ce qui détermine la lentille qui sera implantée en salle d’opération.

## Affichage dans le composant Calculator (Frontend)

Sur l’interface Angular, la section Calculator propose :

Une liste des calculs existants pour un examen.

Un formulaire dynamique pour créer un nouveau calcul.

Des boutons d’action pour sélectionner, modifier ou supprimer un calcul.

Comportements spécifiques :

La lentille sélectionnée est affichée dans un tableau, avec ses paramètres.

L’utilisateur peut voir le nom, la formule, la puissance et le type de lentille.

Un champ visuel indique clairement le calcul sélectionné (badge ou icône).

## Logique Backend

Le service lié au calcul (CalculService) propose notamment :

java

Copier

Modifier

List<Calcul> getCalculsByExamId(Long examId);

Calcul saveCalcul(CalculDto dto);

void selectCalcul(Long calculId);

Ces opérations sont sécurisées et filtrées selon les droits de l’utilisateur.

## Cas d’utilisation typique

Création de calculs

L’ophtalmologue lance l’analyse après un examen biométrique.

Il encode plusieurs formules et associe des lentilles différentes.

Sélection

Une fois validé, un seul calcul est marqué selected = true.

Les autres sont automatiquement désélectionnés.

Exploitation

Ce calcul sera repris dans le Summary et dans les documents de préparation opératoire.

## Intégration avec la synthèse opératoire (Summary)

Lorsqu’un utilisateur consulte le résumé opératoire, l’application récupère :

Le calcul sélectionné (via le filtre isSelected()).

Les détails de la lentille (nom, fabricant, puissance).

D’autres champs seront ajoutés (matériau, constante, etc.) à l’avenir.

java

Copier

Modifier

if (calcul != null && calcul.getLens() != null) {

dto.setSelectedIOL(calcul.getLens().getName());

dto.setManufacturer(calcul.getLens().getManufacturer());

dto.setPower(calcul.getLens().getPower() != null ? calcul.getLens().getPower().toString() : null);

}

## Extension possible (Roadmap)

Des évolutions sont prévues pour enrichir ce module :

Historisation des calculs (log, versioning).

Saisie des mesures réfractives post-opératoires (comparaison avec l’objectif).

Intégration d’un moteur de calcul automatique avec bibliothèques tierces (Barrett Universal, Hill-RBF).

Affichage d’un graphe des prédictions.

# Gestion des antécédents médicaux et ophtalmologiques

## Objectif général

Le module des antécédents médicaux et ophtalmologiques joue un rôle fondamental dans l’évaluation du profil de risque du patient, la stratégie anesthésique et le choix de la technique opératoire.

L'objectif est de permettre aux utilisateurs :

D’encoder, visualiser, modifier et supprimer les antécédents médicaux/ophtalmologiques d’un patient,

De les distinguer selon la latéralité (œil gauche, œil droit, bilatéral),

D’ajouter des remarques libres en cas de cas atypiques.

## Structure des données

Les antécédents sont encodés dans l’entité Exam, directement liée à un patient via un lien Link.

java

Copier

Modifier

private String atcdO; // Antécédents Ophtalmologiques

private String medicalHistory; // Antécédents Médicaux

private String allergy; // Allergies

Ces trois champs sont typiquement utilisés :

Champ Contenu attendu

atcdO Pathologies ophtalmiques antérieures (ex : DMLA, Glaucome)

medicalHistory Pathologies générales (ex : diabète, HTA, antécédents AVC)

allergy Liste d’allergies connues (ex : anesthésiques, antibiotiques)

Format libre : ces champs sont typiquement des chaînes de caractères enrichies ou des listes concaténées.

## Fonctionnement Frontend

Composant Angular associé : SummaryComponent / BiometryComponent

Les champs sont accessibles :

Via des champs textarea modifiables ou en lecture seule,

Dans un formulaire réactif Angular (avec FormBuilder, FormGroup).

Fonctionnalités spécifiques :

Encodage via des zones de texte (libre ou via auto-complétion),

Affichage dynamique dans un onglet dédié ou une carte collapsible,

Gestion de la présence d’"Other" : si l’utilisateur coche une pathologie non listée, un champ texte s’affiche automatiquement pour justification.

Exemple d’implémentation Angular

html

Copier

Modifier

<p-panel header="Antécédents">

<div class="p-fluid">

<label for="medicalHistory">Antécédents médicaux :</label>

<textarea pInputTextarea [(ngModel)]="exam.medicalHistory"></textarea>

<label for="atcdO">Antécédents ophtalmologiques :</label>

<textarea pInputTextarea [(ngModel)]="exam.atcdO"></textarea>

<label for="allergy">Allergies :</label>

<textarea pInputTextarea [(ngModel)]="exam.allergy"></textarea>

</div>

</p-panel>

## Traitement côté backend

Les données sont enregistrées directement via l’objet Exam ou via un DTO intermédiaire.

Exemple dans le service :

java

Copier

Modifier

exam.setAtcdO(dto.getAtcdO());

exam.setMedicalHistory(dto.getMedicalHistory());

exam.setAllergy(dto.getAllergy());

Lors de l’appel de la méthode getSummaryByPatientId(), ces informations sont injectées dans le DTO SummaryDto.

## Particularités UX et cas d’usage métier

Les antécédents sont visibles par tous les rôles, mais seuls les profils médecins ou infirmiers sont autorisés à les modifier.

Un système d’icônes ou de tags peut être utilisé dans le tableau de bord pour signaler la présence d’un antécédent à risque (ex : anticoagulants, glaucome, myopie forte).

Les allergies sont mises en évidence avec une couleur spécifique (ex : fond rouge) en cas de contenu.

## Évolutions possibles

Des évolutions sont prévues pour professionnaliser davantage ce module :

Passage à une structure tabulaire relationnelle avec MedicalHistory et OphthalmicHistory en tant qu’entités à part entière,

Ajout de valeurs suggérées contrôlées (checkbox, dropdown),

Historisation des modifications (audit trail),

Intégration de règles cliniques automatisées (alertes en cas d’interactions).

# Gestion du côté dominant et de la latéralité

## Objectif clinique et fonctionnel

La latéralité (œil gauche, œil droit) et le côté dominant (visuellement parlant) sont deux éléments clés pour :

Planifier correctement la chirurgie (s'assurer qu'on opère le bon œil),

Adapter le choix de la lentille en fonction de la dominance visuelle,

Tenir compte de l'œil préférentiel du patient dans la stratégie post-opératoire (ex : monovision).

## Données concernées dans l’entité Exam

L’ensemble de ces données sont gérées dans l’entité Exam.java :

java

Copier

Modifier

@Enumerated(EnumType.STRING)

private EyeSide eyeSide;

private String dominantEye;

EyeSide (latéralité de l’examen / chirurgie)

Il s’agit d’un enum personnalisé, généralement défini ainsi :

java

Copier

Modifier

public enum EyeSide {

OD, // Oeil Droit

OS, // Oeil Gauche

OU // Bilatéral

}

Cela permet une gestion typée et rigoureuse côté backend comme frontend.

DominantEye (œil dominant)

Encodé comme une chaîne de caractères (libre ou via enum/constantes). Valeurs attendues :

"OD" (œil droit)

"OS" (œil gauche)

"Indéterminé" ou "Non précisé" (en l’absence de dominance claire)

## Gestion côté frontend (Angular)

Formulaire de saisie / modification

Le composant Angular concerné (ex : ExamFormComponent, BiometryComponent) affiche des champs spécifiques :

html

Copier

Modifier

<label for="eyeSide">Latéralité :</label>

<p-dropdown

[options]="eyeSideOptions"

[(ngModel)]="exam.eyeSide"

placeholder="Choisir l'œil">

</p-dropdown>

<label for="dominantEye">Œil dominant :</label>

<p-dropdown

[options]="dominantEyeOptions"

[(ngModel)]="exam.dominantEye"

placeholder="Choisir la dominance">

</p-dropdown>

Comportement dynamique

Si le champ eyeSide est modifié, des validations spécifiques peuvent s’appliquer :

Exemple : interdire OU (bilatéral) si unilatéral obligatoire.

Le champ dominantEye peut être :

Affiché en suggestion,

Déduit via une logique métier (ex : dominance statistique),

Modifiable à tout moment par l’utilisateur.

## Exploitation dans le DTO résumé (SummaryDto)

La méthode getSummaryByPatientId(Long patientId) alimente le SummaryDto :

java

Copier

Modifier

dto.setLaterality(selectedExam.getEyeSide().name());

dto.setDominantEye(selectedExam.getDominantEye());

Ces valeurs sont utilisées :

Pour l’affichage dans l’interface médecin,

Pour générer des rapports PDF,

Pour orienter les choix de calculs biométriques et d’implants (selon dominance).

## Intégration dans les workflows chirurgicaux

Avantages cliniques de cette gestion rigoureuse :

Réduction du risque opératoire (éviter l’erreur de côté),

Meilleure planification de la vision post-opératoire (ex : monovision avec correction d’un œil pour la lecture),

Communication claire avec l’équipe opératoire.

## Évolutions envisagées

Remplacer la string dominantEye par un enum Dominance pour fiabiliser les valeurs,

Ajouter une vérification croisée si dominantEye ≠ eyeSide avec une alerte (utile pour chirurgie volontaire du non-dominant),

Intégration dans le module décisionnel de calcul d’implant (via influence sur la cible réfractive),

Stockage dans une table dédiée si gestion longitudinalement répétée.

# Gestion des implants intraoculaires (IOLs)

## Contexte médical et finalité logicielle

Dans la chirurgie de la cataracte, le cristallin opacifié est retiré, puis remplacé par une lentille intraoculaire (IOL). Le choix de l’implant repose sur :

Des mesures biométriques précises (kératométrie, AL, ACD...),

Un modèle de calcul ophtalmologique (Haigis, Barrett, SRK/T...),

Le profil du patient (monovision, presbytie, myopie...).

L'application EyeWeb prend en charge tout ce workflow via une architecture intégrée et modulable.

## Entités concernées

### Entité Lens.java (implant)

C’est la représentation métier de l’implant. Elle comprend des données essentielles pour le calcul, la traçabilité, et la prescription :

java

Copier

Modifier

@Entity

@Table(name = "lens")

public class Lens extends IdentifiedModel {

private String name; // Nom commercial de l’implant (ex: Alcon AcrySof IQ SN60WF)

private String manufacturer; // Fabricant (Alcon, Zeiss, Hoya, etc.)

private Float power; // Puissance optique (en dioptries)

private Float diameter; // Diamètre de l’optique (mm)

private Float aConstant; // Constante A (modèle SRK/T)

private Float sf; // SF (Haigis)

private Float a0; // A0 (Haigis)

private Float a1; // A1 (Haigis)

private Float a2; // A2 (Haigis)

@Enumerated(EnumType.STRING)

private LensType type; // MONOFOCAL, MULTIFOCAL, TORIC, EDOF...

private String reference; // Référence fabricant

private String udi; // Identifiant unique (Unique Device Identifier)

private boolean active; // Actif ou obsolète (filtrage en interface)

}

### Enumération LensType

Elle catégorise les implants pour filtrage ou suggestions :

java

Copier

Modifier

public enum LensType {

MONOFOCAL, MULTIFOCAL, TORIC, EDOF, PHASIC

}

### Entité Calcul.java

Elle relie un exam à un implant suggéré ou choisi, en y associant des détails de calcul :

java

Copier

Modifier

@ManyToOne

@JoinColumn(name = "lens\_id")

private Lens lens;

private boolean selected; // Détermine si ce calcul est celui retenu

private String formulaUsed; // Nom de la formule appliquée (Haigis, SRK/T, etc.)

private Float predictedRefraction;

## Relations entre Exam, Calcul et Lens

Un Exam peut générer plusieurs Calcul (ex : simulation de plusieurs formules).

Chaque Calcul peut référencer une et une seule Lens.

Une Lens peut être réutilisée dans plusieurs Calcul.

Représentation logique :

scss

Copier

Modifier

Exam (1) ──── (0..n) Calcul (n) ──── (1) Lens

## Exploitation dans le backend

Sélection de l’implant

La méthode dans CataractServiceImpl ou équivalent utilise :

java

Copier

Modifier

Calcul selectedCalc = exam.getCalculs().stream()

.filter(Calcul::isSelected)

.findFirst()

.orElse(null);

if (selectedCalc != null && selectedCalc.getLens() != null) {

dto.setSelectedIOL(selectedCalc.getLens().getName());

dto.setManufacturer(selectedCalc.getLens().getManufacturer());

dto.setPower(selectedCalc.getLens().getPower().toString());

}

API REST typique

GET /api/lenses → liste des implants disponibles

GET /api/lenses/{id} → détails d’un implant

POST /api/calculs → créer un calcul basé sur l’examen + formule

PUT /api/calculs/{id}/select → marquer un calcul comme « sélectionné »

## Exploitation dans le frontend

Composant : LensSelectorComponent

Utilise p-dropdown ou p-autocomplete (PrimeNG) pour suggérer un implant filtré selon :

Le type de lentille souhaité,

Le fabricant,

La plage de puissance.

html

Copier

Modifier

<p-dropdown

[options]="lensList"

[(ngModel)]="selectedLens"

optionLabel="name"

placeholder="Sélectionner un implant">

</p-dropdown>

Tableau comparatif

Affiche plusieurs calculs avec différentes lentilles, avec colonnes :

Lentille (nom + fabricant),

Formule utilisée,

Prédiction réfractive,

Sélection (radio bouton)

## Sécurité et audit

Chaque sélection de lentille est enregistrée avec trace d’utilisateur (via audit ou token).

Les implants obsolètes (active = false) sont cachés au frontend.

## Évolutions possibles

Intégration d’un moteur de calcul IOL personnalisé (algorithme maison).

Ajout de la gestion des toricités (axe, puissance cylindrique).

Historique complet des implants posés chez un patient (multi-yeux / multi-séjours).

Suggestions intelligentes via un moteur d’aide à la décision (basé sur l’IA ou Règles métier).

# Calculs biométriques et formules ophtalmiques

## Objectifs cliniques et fonctionnels

Les calculs biométriques sont au cœur de la planification chirurgicale de la cataracte. Ils permettent, à partir de mesures anatomiques précises de l'œil, de prédire la puissance de l’implant intraoculaire nécessaire pour corriger la vision.

L'application EyeWeb automatise ce processus, en exploitant :

Les données mesurées lors de l’examen (biométrie),

Les formules ophtalmiques standardisées (SRK/T, Haigis, Barrett Universal II, etc.),

Les données associées aux implants disponibles.

## Données biométriques traitées

Les champs biométriques pertinents sont intégrés à l'entité Exam. Parmi les plus utilisés :

Donnée Colonne Exam Description

Longueur axiale (AL) al Distance du pôle antérieur à la rétine

Profondeur de chambre acd, internalAcd Distance entre la cornée et le cristallin

Courbure cornéenne k1, k2, k1Axis, k2Axis Kératométrie

Diamètre pupillaire pupilDia, pupilMin, pupilMax Taille pupille

Épaisseur lentille lensThickness Indice clé pour Haigis

CCT (épaisseur cornée) cct Corrélé à l’astigmatisme et fiabilité des mesures

WTW (blanc à blanc) wtw Largeur de la cornée mesurée horizontalement

Ces champs sont initialisés automatiquement lors de l'import d'un examen depuis une machine biométrique (Zeiss, IOLMaster, etc.) via l'entité Exam.

## Entité Calcul et prédiction

Chaque examen peut produire plusieurs Calcul contenant :

* Une formule de calcul choisie (champ formulaUsed)
* Une lentille associée (champ lens)
* Une réfraction prédite (champ predictedRefraction)
* Un indicateur de sélection (selected)

@Entity

@Table(name = "calcul")

public class Calcul extends IdentifiedModel {

private String formulaUsed; // Ex: "SRK/T", "Haigis"

private Float predictedRefraction;

@ManyToOne

@JoinColumn(name = "lens\_id")

private Lens lens;

@ManyToOne

@JoinColumn(name = "exam\_id")

private Exam exam;

private boolean selected;

}

## Fonctionnement d’un calcul

Le calcul peut être déclenché via :

* Le backend (appel API REST : /calculs)
* Une interaction utilisateur dans le frontend (sélection manuelle ou auto-suggestion)

Le service métier applique alors la formule sélectionnée. Voici une simplification du workflow :

Float AL = exam.getAl();

Float ACD = exam.getAcd();

Float K1 = exam.getK1();

Float K2 = exam.getK2();

Float Kavg = (K1 + K2) / 2;

Float Aconstant = lens.getAConstant();

Float predictedPower = (1336 / (AL - 0.05)) - (Kavg); // SRK/T simplifié

Calcul c = new Calcul();

c.setFormulaUsed("SRK/T");

c.setPredictedRefraction(predictedPower - lens.getPower());

c.setSelected(false);

## Sélection du meilleur calcul

Le Calcul sélectionné est celui :

* Le plus proche de la réfraction cible (targetRefrSph)
* Associé à un implant disponible
* Compatible avec les contraintes du patient
* Ce calcul est marqué via calcul.setSelected(true).
* L'IOL retenue s'affiche ensuite dans :
* Le composant Summary côté frontend
* Le SummaryDto côté backend

## Services métier associés

CalculService :

* Génération de plusieurs calculs à partir d’un examen
* Application de formules ophtalmiques
* Sélection automatique du meilleur calcul

CataractService :

* Intègre les calculs dans le parcours patient
* Fournit les valeurs à la couche Summary

## Interface utilisateur (Angular)

Le module Calculator affiche :

|  |  |  |
| --- | --- | --- |
| Élément | Composant PrimeNG utilisé | Description |
| Liste des calculs | p-table | Un calcul par ligne, avec IOL, formule, prédiction |
| Sélection IOL manuelle | p-dropdown | Liste des implants filtrés |
| Résultat sélectionné | p-card, p-tag | Affichage mis en valeur |

Interaction :

* L’utilisateur peut sélectionner manuellement un calcul.
* Le calcul est alors mis à jour en BDD et marqué comme selected.

## Sécurité et audit

* Chaque modification de calcul est auditée
* Seuls les utilisateurs autorisés (chirurgien, assistant) peuvent modifier les choix d'IOL
* Historique des modifications possible via Envers (Audited)

## Perspectives d’évolution

* Intégration de l'algorithme Barrett Universal II (API payante ou implémentation propre)
* Apprentissage automatique basé sur les cas historiques pour ajuster les prédictions
* Calcul en batch lors de l’import de mesures (pré-chargement)
* Gestion des contraintes cornéennes spécifiques (astigmatisme, toricité)

# Conclusion

La mise en œuvre de l'application EyeWeb représente l’aboutissement d’un projet à la fois ambitieux et réaliste, conjuguant exigences cliniques, rigueur technique et finesse d’architecture logicielle. Le développement de cette solution a nécessité la mobilisation d’un large éventail de compétences, allant de la modélisation des données biométriques jusqu’à l’implémentation sécurisée des interfaces utilisateur, en passant par la gestion des calculs de lentilles, des rôles, des autorisations et du parcours opératoire.

D’un point de vue technique, EyeWeb s’appuie sur un écosystème solide : Java 17, Spring Boot, JPA, JWT, PostgreSQL, Angular, PrimeNG... autant de briques technologiques éprouvées, judicieusement agencées dans une architecture modulaire et maintenable, qui autorise aussi bien l’extension fonctionnelle que l’industrialisation. Le choix de technologies modernes n’a pas été dicté par la tendance, mais bien par leur adéquation aux besoins spécifiques du domaine médical : fiabilité, robustesse, auditabilité, sécurité, et lisibilité.

Au-delà de la simple couverture fonctionnelle, EyeWeb illustre comment une architecture logicielle bien pensée peut contribuer à améliorer la qualité des soins et à réduire les erreurs cliniques dans le domaine de la chirurgie ophtalmique. L'automatisation des calculs, la traçabilité des actions, l’organisation des données patient, la gestion sécurisée des rôles, ou encore l’interface intuitive pensée pour les praticiens, participent à faire de cette plateforme bien plus qu’une application métier : un outil médical intelligent.

Ce travail constitue une base solide pour l’avenir. Il pourrait être enrichi par l’intégration d’algorithmes de prédiction plus complexes, l’interconnexion avec des systèmes hospitaliers (HL7/FHIR), ou l’implémentation de dashboards cliniques. Il est également adaptable à d'autres spécialités médicales nécessitant un suivi pré et post-opératoire rigoureux.

En somme, EyeWeb démontre que la convergence entre ingénierie logicielle et médecine peut donner naissance à des outils puissants, au service de l’expertise humaine, de la décision médicale, et, in fine, de la qualité de vie des patients.