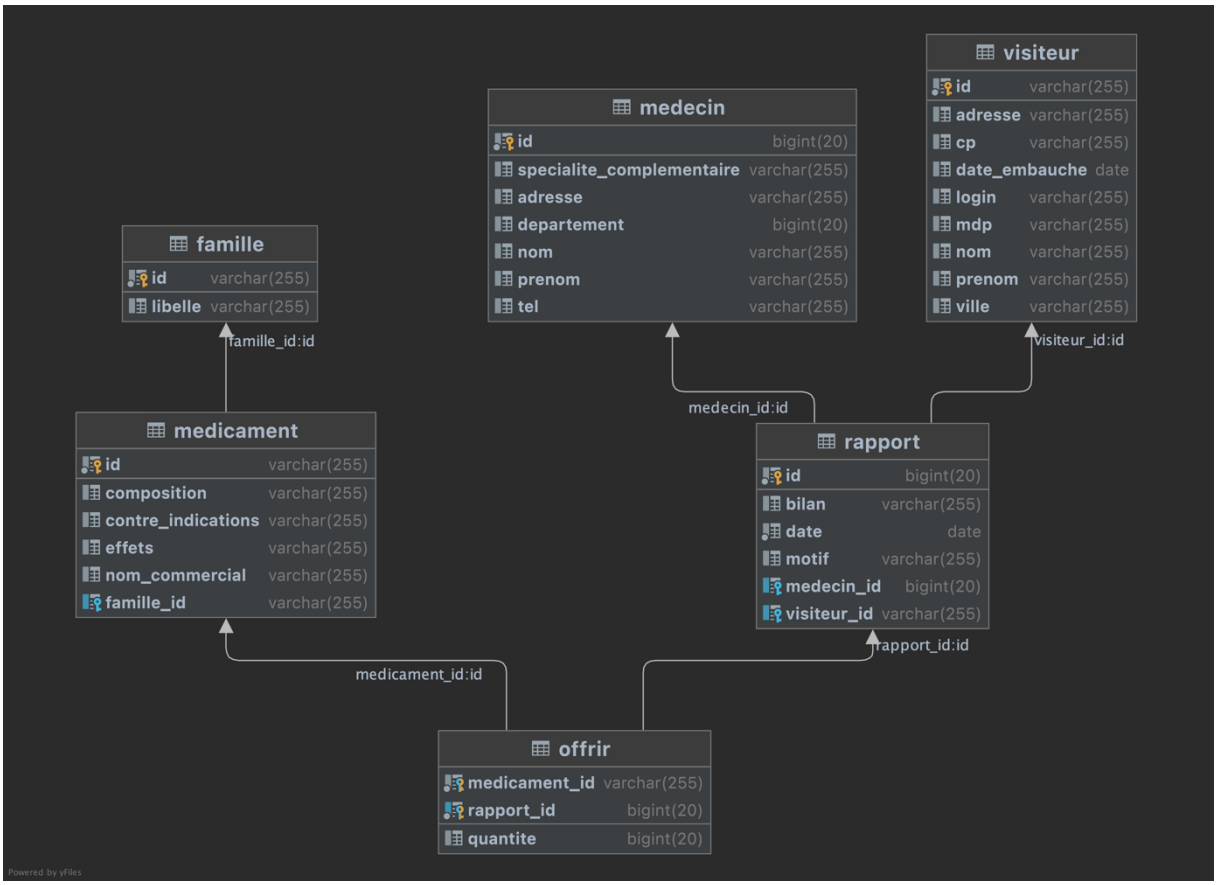
# GSB Gestion Visites

Documentation technique

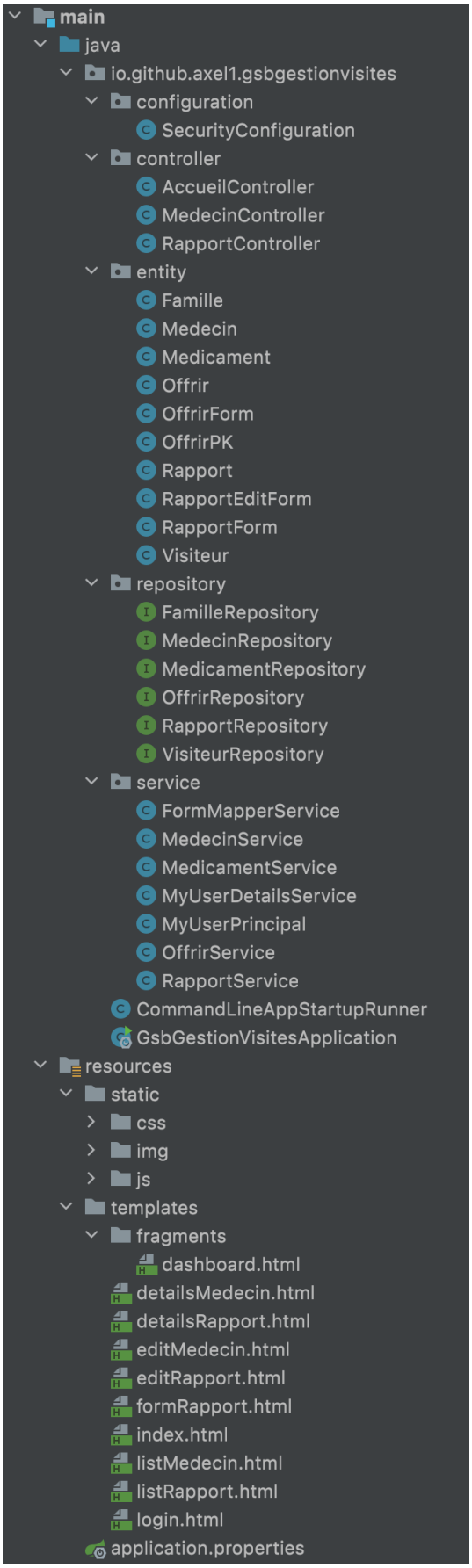
## Introduction

GSB Gestion Visites a été conçu avec Spring Boot et utilise la technologie JVM. Le moteur de base de données utilisé est MariaDB.
Pour télécharger les sources et compiler le projet, veuillez suivre les instructions indiquées sur le dépôt GitHub.

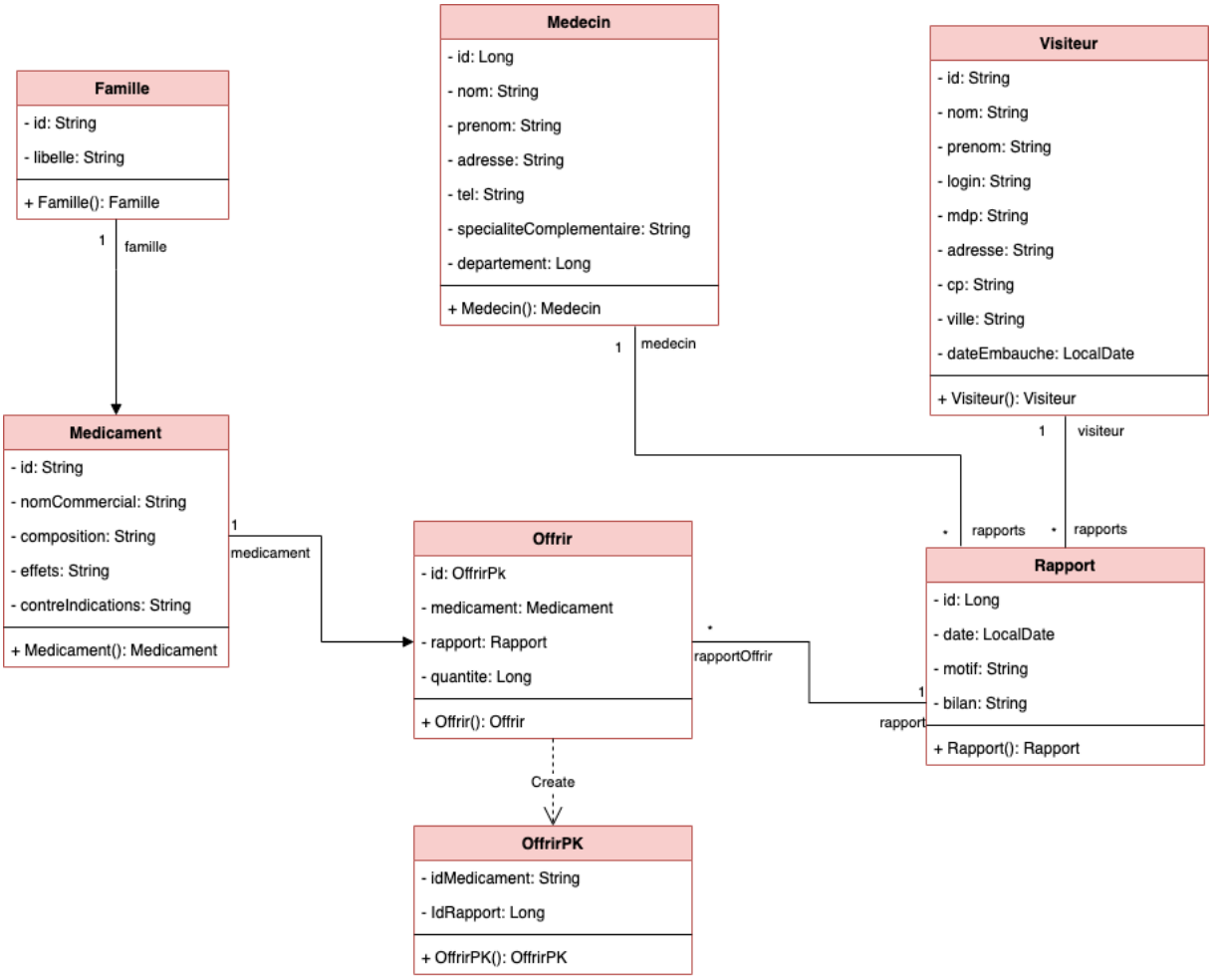
## Structure des données



# Structure de l’application



# Diagramme de classe



# Présentation des classes

## Les Entities

Ce sont les objets métier de l'application, la plupart représentent une table dans la base de données.

```java
@Entity
@Table
public class Rapport {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @NotNull
    @DateTimeFormat(pattern = "yyyy-MM-dd")
    private LocalDate date;
    @NotBlank
    private String motif;
    @NotBlank
    private String bilan;

    @ManyToOne
    private Visiteur visiteur;
    @ManyToOne
    private Medecin medecin;

    @OneToMany(mappedBy = "rapport", fetch = FetchType.EAGER)
    private Set<Offrir> rapportOffrir;
```

On utilise des annotations Java pour configurer les entities. Ainsi, l'ORM (Hibernate) se chargera de générer la structure de la base de données et gérer l'accès aux données à l'aide des repositories.
Les annotations tel que @NotNull @DateTimeFormat etc… permettent d'assurer la validation de l'objet et de retourner les éventuelles erreurs.
Les annotations tel que @ManyToOne @OneToMany etc… permettent de définir les relations entre les entités et de générer les clés étrangères dans la base de données.

## Les Repositories

```java
@Repository
public interface RapportRepository extends JpaRepository<Rapport, Long> {
    List<Rapport> findByVisiteur(Visiteur visiteur);

    Optional<Rapport> findByVisiteurAndId(Visiteur visiteur, Long id);

    List<Rapport> findByVisiteurAndDate(Visiteur visiteur, LocalDate date);
}
```

Les repositories permettent l'accès aux données c'est-à-dire charger et décharger les objets depuis la base de données.
Ce sont des interfaces qui seront implémentées à la volée par le framework.
Pour créer des requêtes personnalisées, il suffit de définir la signature de la méthode.
Attention cependant, le nom de la méthode est important car il permet à Spring de générer la requête ainsi que le code correspondant.

## Les Services

Les classes services permettent d'effectuer la logique métier. Ces classes font la liaison entre les repositories et les controllers. En effet, le controller ne doit pas accéder directement aux repositories, il doit d'abord passer par une classe service.

```java
@Service
public class MedicamentService {
    private final MedicamentRepository medicamentRepository;

    @Autowired
    public MedicamentService(MedicamentRepository medicamentRepository) {
        this.medicamentRepository = medicamentRepository;
    }

    public List<Medicament> getAllMedicament() {
        return medicamentRepository.findAll();
    }

    public Optional<Medicament> findMedicamentById(String id) {
        return medicamentRepository.findById(id);
    }
}
```

Dans cet exemple, la classe service n'effectue aucune logique métier. Elle fait seulement abstraction de la repository pour le controller.

## Les Controllers

```java
@Controller
@RequestMapping("/rapports")
public class RapportController {
    private final RapportService rapportService;
    private final MedecinService medecinService;
    private final MedicamentService medicamentService;
    private final FormMapperService formMapperService;
    private final OffrirService offrirService;

    @Autowired
    public RapportController(RapportService rapportService, MedecinService medecinService, MedicamentService medicamentService,
        this.rapportService = rapportService;
        this.medecinService = medecinService;
        this.medicamentService = medicamentService;
        this.formMapperService = formMapperService;
        this.offrirService = offrirService;
    }

    @GetMapping(path = "")
    public String searchRapport(Model model, Authentication authentication, @RequestParam(defaultValue = "") String date) {
        MyUserPrincipal myUserPrincipal = (MyUserPrincipal) authentication.getPrincipal();
        Visiteur visiteur = myUserPrincipal.getVisiteur();
        if (Objects.equals(date, b: "")) {
            model.addAttribute( attributeName: "title", attributeValue: "Rapports");
            model.addAttribute( attributeName: "rapports", rapportService.getRapportByVisiteur(visiteur));
            return "listRapport";
        } else {
            LocalDate localDate = LocalDate.parse(date);
            model.addAttribute( attributeName: "title", attributeValue: "Rapports");
            model.addAttribute( attributeName: "rapports", rapportService.getRapportByVisiteurAndDate(visiteur, localDate));
            return "listRapport";
        }
    }

    @GetMapping("/{id}")
    public String getRapportById(Model model, @PathVariable("id") Long id, Authentication authentication) {
        MyUserPrincipal myUserPrincipal = (MyUserPrincipal) authentication.getPrincipal();
        Visiteur visiteur = myUserPrincipal.getVisiteur();

        Optional<Rapport> rapportOptional = rapportService.findRapportByVisiteurAndId(visiteur, id);

        if (rapportOptional.isPresent()) {
            Rapport rapport = rapportOptional.get();

            model.addAttribute( attributeName: "title", attributeValue: "Rapports / Détails");
            model.addAttribute( attributeName: "rapport", rapport);

            return "detailsRapport";
        } else {
            throw new ResponseStatusException(HttpStatus.NOT_FOUND);
        }
    }
}
```

Les controllers permettent de définir les endpoints de notre application Web, ainsi qu'envoyer et recevoir les données des templates (vues en html). Le controller appelle ensuite les services pour charger et enregistrer les données.
Les annotations @RequestMapping permettent de définir les routes ainsi que les verbes HTTP (GET, POST etc…).
Les méthodes acceptent en paramètres des objets tel que Model ou Authentication, en effet ces objets seront injectés à la volée par le framework.
L'objet Model permet de définir les attributs envoyés à la vue. L'objet Authenticaiton permet de récupérer des informations en rapport avec l'authentification.

## Les Forms

Ce sont des classes techniques qui permettent de faciliter la création et la validation de formulaire.

```java
public class RapportForm {
    @Valid
    private Rapport rapport;
    @NotNull
    private Long medecinId;
    @Valid
    private List<OffrirForm> offrirForms = new ArrayList<>();
```

Le moteur de template thymeleaf permet de créer des formulaires à partir d'un objet, dans le cas d'un formulaire complexe qui ne correspond pas exactement aux propriétés de l'objet métier, il faut créer un objet wrapper qui contiendra plusieurs objets ou plusieurs champs pour correspondre avec le formulaire.

```java
    @GetMapping("/new")
    public String createRapport(Model model) {
        RapportForm rapportForm = new RapportForm();
        rapportForm.setRapport(new Rapport());
        List<Medecin> medecinList = medecinService.getAllMedecin();
        List<Medicament> medicamentList = medicamentService.getAllMedicament();
        model.addAttribute( attributeName: "title",  attributeValue: "Rapports / Nouveau");
        model.addAttribute( attributeName: "rapportForm", rapportForm);
        model.addAttribute( attributeName: "medecinList", medecinList);
        model.addAttribute( attributeName: "medicamentList", medicamentList);
        return "formRapport";
    }
```

Le controller crée un RapportForm vide, ajoute un nouveau Rapport vide à l'intérieur puis envoie le RapportForm à la vue.

```
<form class="row g-3" method="POST" th:action="@{/rapports}"
      th:object="${rapportForm}">
    <div class="alert alert-danger" role="alert" th:if="${#fields.hasAnyErrors()}">
        Une erreur est survenue, veuillez vérifier votre saisie
    </div>

    <div class="col-md-6">
        <label class="form-label">ID</label>
        <input class="form-control" disabled name="inputRapportID"
               th:value="${rapportForm.getRapport().id}" type="text">
    </div>
    <div class="col-md-6">
        <label class="form-label" for="inputRapportDate">Date</label>
        <input class="form-control" id="inputRapportDate" name="inputRapportDate"
               th:errorclass="is-invalid"
               th:field="*{rapport.date}" th:value="${rapportForm.getRapport().date}"
               type="date">
        <div class="invalid-feedback" th:if="${#fields.hasErrors('rapport.date')}">
            Veuillez saisir une date valide
        </div>
    </div>
    <div class="col-md-6">
        <label class="form-label" for="inputRapportMedecin">Medecin</label>
        <input class="form-control" id="inputRapportMedecin" list="datalistOptions"
               name="inputRapportMedecin" th:errorclass="is-invalid" th:field="*{medecinId}">
        <datalist id="datalistOptions">
            <option th:each="medecin : ${medecinList}"
                    th:text="${medecin.getNom()} + ' ' + ${medecin.getPrenom()}"
                    th:value="${medecin.getId()}"></option>
        </datalist>
        <div class="invalid-feedback" th:if="${#fields.hasErrors('medecinId')}">
            Veuillez choisir un médecin
        </div>
    </div>
```

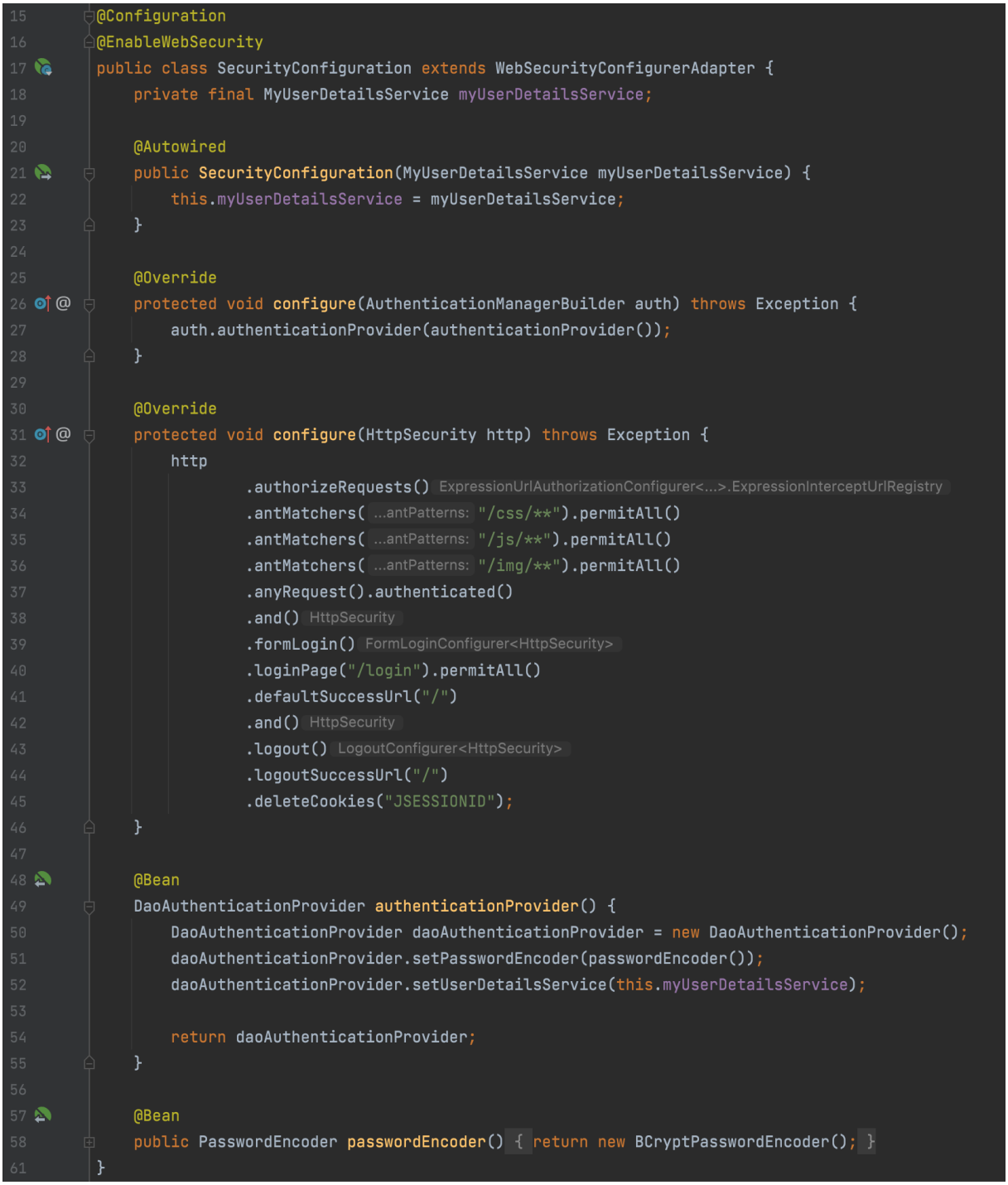
On récupère ensuite l'objet FormRapport complété par le formulaire de la vue dans le controller puis on utilise la classe FormMapperService pour unwrap l'objet FormRapport vers un objet Rapport complet qui pourra être sauvegardé dans la base de données.

```
26 @    public Rapport toRapport(RapportForm rapportForm, Visiteur visiteur) throws Exception {
27          Rapport rapport = rapportForm.getRapport();
28          Optional<Medecin> medecinOptional = medecinService.findMedecinById(rapportForm.getMedecinId());
29
30          if (medecinOptional.isPresent()) {
31              Medecin medecin = medecinOptional.get();
32              rapport.setMedecin(medecin);
33              rapport.setVisiteur(visiteur);
34              return rapport;
35          } else {
36              throw new Exception();
37          }
38      }
```

Cette méthode se charge également de lier l'objet Rapport avec l'objet Visiteur.

# Authentification

Cette application utilise Spring Security pour gérer l'authentification.



```
@Configuration
@EnableWebSecurity
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {
    private final MyUserDetailsService myUserDetailsService;

    @Autowired
    public SecurityConfiguration(MyUserDetailsService myUserDetailsService) {
        this.myUserDetailsService = myUserDetailsService;
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.authenticationProvider(authenticationProvider());
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
                .authorizeRequests() ExpressionUrlAuthorizationConfigurer<...>.ExpressionInterceptUrlRegistry
                .antMatchers( ...antPatterns: "/css/**").permitAll()
                .antMatchers( ...antPatterns: "/js/**").permitAll()
                .antMatchers( ...antPatterns: "/img/**").permitAll()
                .anyRequest().authenticated()
                .and() HttpSecurity
                .formLogin() FormLoginConfigurer<HttpSecurity>
                .loginPage("/login").permitAll()
                .defaultSuccessUrl("/")
                .and() HttpSecurity
                .logout() LogoutConfigurer<HttpSecurity>
                .logoutSuccessUrl("/")
                .deleteCookies("JSESSIONID");
    }

    @Bean
    DaoAuthenticationProvider authenticationProvider() {
        DaoAuthenticationProvider daoAuthenticationProvider = new DaoAuthenticationProvider();
        daoAuthenticationProvider.setPasswordEncoder(passwordEncoder());
        daoAuthenticationProvider.setUserDetailsService(this.myUserDetailsService);

        return daoAuthenticationProvider;
    }

    @Bean
    public PasswordEncoder passwordEncoder() { return new BCryptPasswordEncoder(); }
}
```

La configuration des pages accessibles se fait dans la méthode configure en appelant les méthodes « antMatchers » sur l'objet « http ». D'autres configurations sont également possible tel que la page de connexion ou la page de redirection après la déconnexion.

# Configuration

## Base de données

La configuration de la base de données s'effectue dans le fichier application.properties

```
spring.datasource.url=jdbc:mariadb://10.211.55.11:3306/GSBVisites
spring.datasource.username=util
spring.datasource.password=util
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MariaDBDialect
```

Spring Boot supporte un grand nombre de SGBDR, vous pouvez donc utiliser la plupart des moteurs de base de données en configurant le dialect adéquat.

## Motifs

Vous pouvez configurer les motifs disponibles dans le formulaire de création et d'édition de rapport dans la classe MotifConfiguration

```
@Configuration
public class MotifConfiguration {
    private final List<String> motifs = new ArrayList<>();

    public MotifConfiguration() {
        motifs.add("Demande du médecin");
        motifs.add("Recommandation de confrère");
        motifs.add("Installation nouvelle");
        motifs.add("Visite annuelle");
        motifs.add("Prise de contact");
        motifs.add("Conseil d'un collègue");
        motifs.add("Nouveau médecin, prise de contact");
    }

    public List<String> getMotifs() { return motifs; }

    public boolean validate(Rapport rapport) { return this.motifs.contains(rapport.getMotif()); }
}
```