Bloc de construction de l'application de la PoC

Développement d'une Preuve de Concept

MedHead

Auteur(s) et contributeur(s)

Nom & Coordonnées	Qualité & Rôle	Société
Gérald ATTARD	Consultant en Architecture logicielle	Xxxxxxxxxx

Historique des modifications et des révisions

N° version	Date	Description et circonstance de la modification	Auteur
1.0	18/01/2023	Création du document	Gérald ATTARD

Validation

N° version	Nom & Qualité	Date & Signature	Commentaires & Réserves
1.0	Kara Trace CIO, Ursa Major Health		
	Anika Hansen, PDG, Jupiter Scheduling Inc.		
	Chris Pike Architecte d'entreprise principal, Schedule Shed		

Tableau des abréviations

Abr.	Sémantique
CLUF	Contrant de Licence Utilisateur Final
MTBF	Mean-Time-Between-Failure (temps moyen qui sépare deux erreurs en production)
MTTR	Mean-Time-To-Recover (temps moyen de correction entre deux erreurs de production)
MSA	MicroServices Architecture (trad. *architecture en micro-services*)
MVP	Minimum Viable Product (trad. *produit minimal viable*)
Repo	Repository (trad. *dépôt de fichiers*)
SAW	Statement for Architecture Work (trad. *déclaration pour l'architecture*)
SBB	Solution Building Block (trad. *bloc de construction de la solution*)

Table des matières

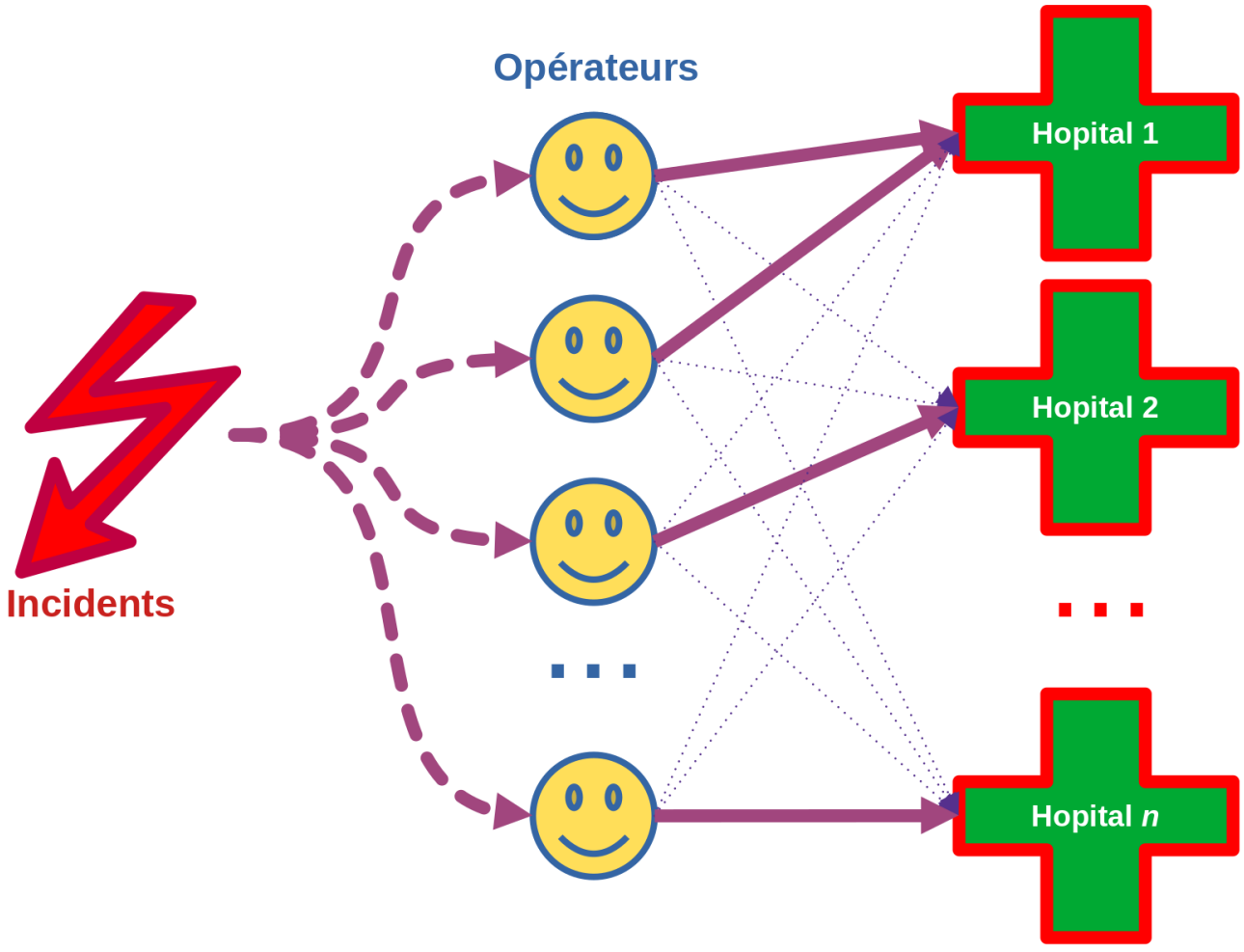
I. Introduction

Cette SBB a pour but de présenter l'application mettant en œuvre la PoC de prise en compte des messages d'urgence pour le Consortium MedHead.

Tel qu'il a été introduit dans les documents *SAW* et *plan de tests*, cette PoC sera structurée autour de trois ressources principales :

- les incidents survenus ;
- les hopitaux devant prendre en charge les incidents ;
- les opérateurs chargés de répondre aux messages d'urgence et de les répartir dans les différents hôpitaux.

Ainsi, les intéractions de ces ressources seront représentées seront structurées comme suit :



II. Vue d'ensemble

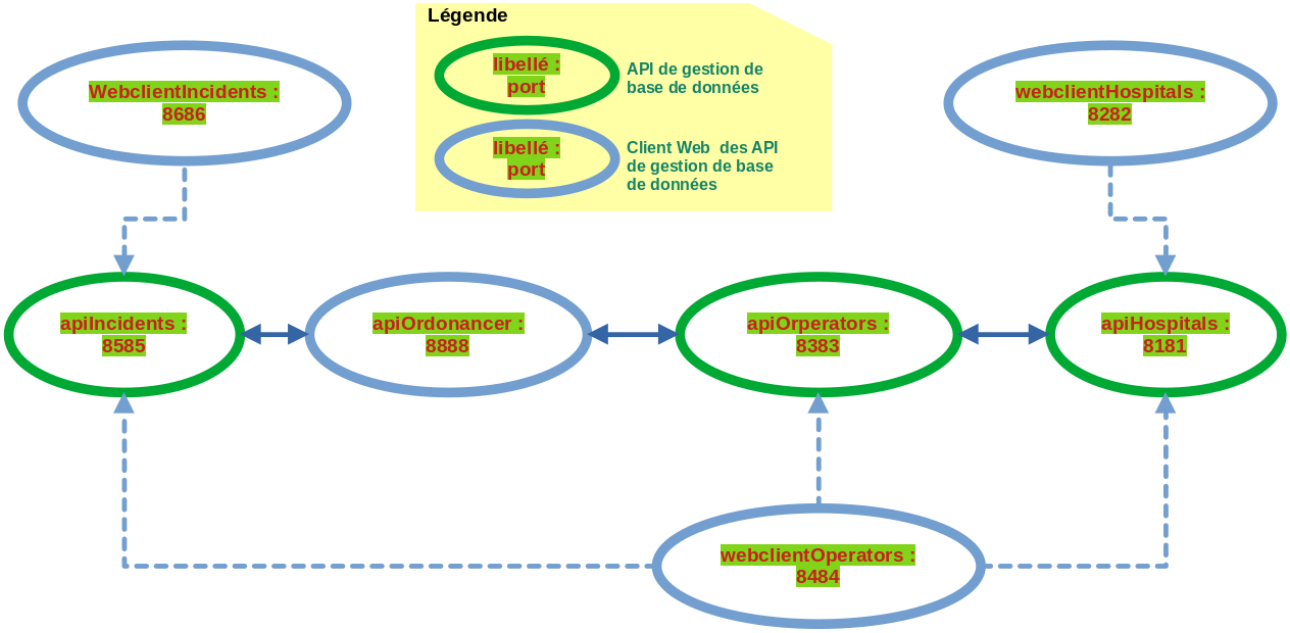
Relativement à ce qui a été présenté en introduction, il est alors possible de dénombrer quatre composants essentiels au sein de la PoC, tels que :

- Composant *apiIncidents* : ce composant sera chargé de simuler la génération d'incidents.
- Composant *apiOperators* : ce composant aura pour objectif de déclarer et gérer les opérateurs chargés de prendre en compte les incidents et de laes répartir vers les hôpitaux adéquats.
- Composant *apiHospitals* : ce composant recensera les différents hôpitaux ainsi que leurs attributs, tels que le nombre de lots au total, le nombre de lits disponibles, leurs spécialités médicales présentes…
- Composant *apiOrdonnancer* : ce composant sera chargé d'attribuer les incidents aux opérateurs en fonction de leur disponibilité.

En outre, aux trois premiers composants recensés ci-dessus, la PoC proposera un client web permettant d'afficher les bases de données de chacun, à savoir :

- Composant *webclientIncidents* : le composant web permettant d'afficher la base de données des différents incidents.
- Composant *webclientOperators* : le composant web permettant d'afficher la base de données des différents opérateurs.
- Composant *webclientHospitals* : le composant web permettant d'afficher la base de données des différents hôpitaux.

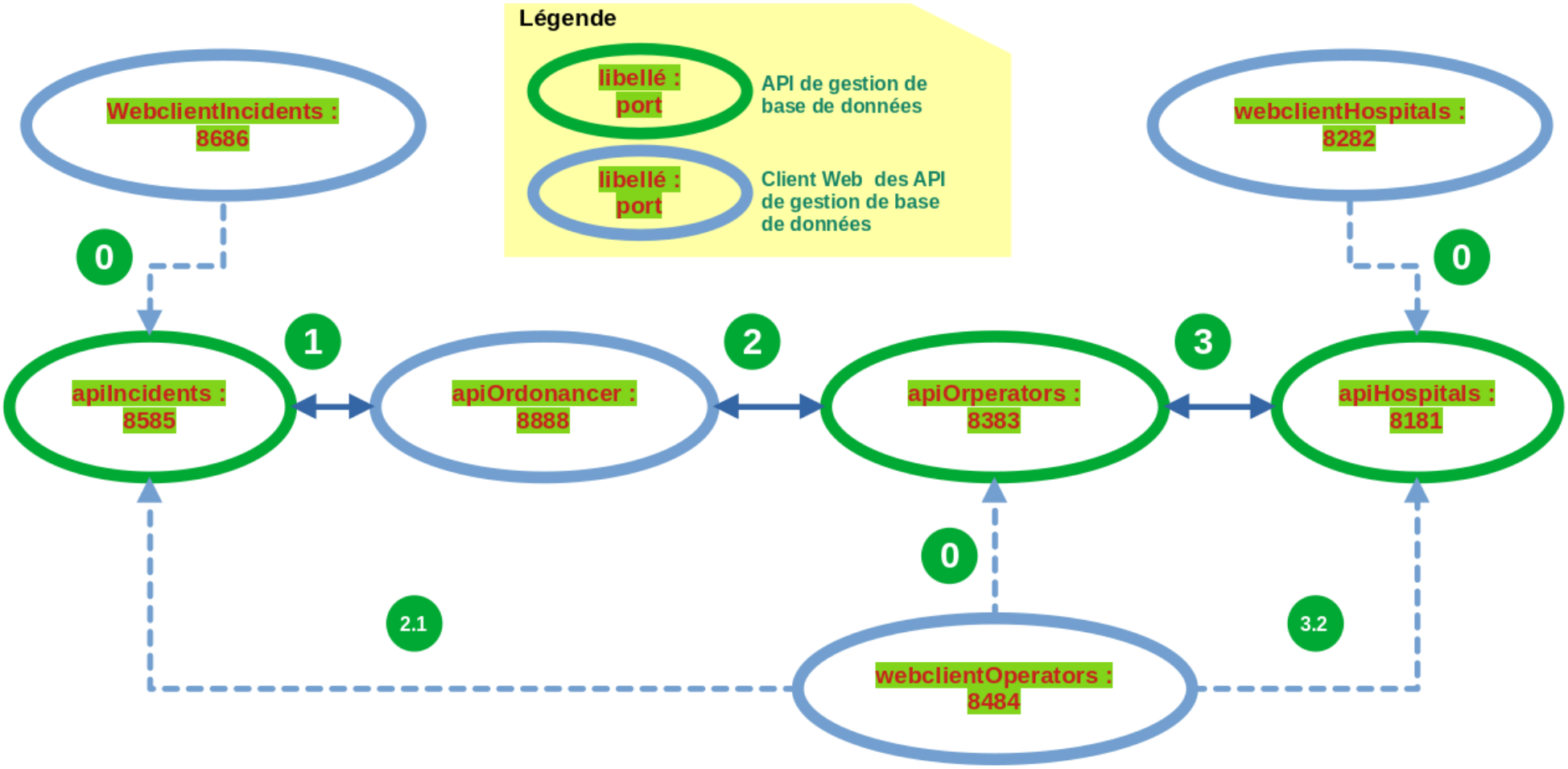
Ainsi, ces différents composants se structuraont selon le diagramme ci-dessous :



En se basant sur le diagramme ci-dessus, il est possible d’ordonnancer les actions réalisées par la PoC :

- Etapes 0 d’initialisation :
  - étape 0 : des hôpitaux sont existants et il est possible de visualiser leurs attributs via le client web ;
  - étape 0 : des opérateurs sont existants et il est possible de visualiser leurs attributs via le client web ;
  - étape 0 : des incidents surviennent et il est possible de visualiser leurs attributs via le client web ;
- Etapes 1 et 2 de traitements (actions de l'ordonnanceur 1 et des opérateurs 2) :
  - étape 1.1 l'ordonnanceur identifie l'incident en cours de traitement (traitement à "En cours") ;
  - étape 1.2 l'ordonnanceur cherche un opérateur disponible et va patienter tant qu'il n'y en aura pas de disponible
  - étape 1.3 l'ordonnanceur attribue l'incident à l'opérateur "disponible"
  - étape 1.4 l'opérateur prend en charge l'incident en devenant "Non disponible"
  - étape 1.5 boucle sur l'étape 1.1
  - étape 2.1 l'opérateur va chercher un hopital correspondant aux critères de l'incident (critères de spécialité médicale, localisation, lits disponibles) ;
  - étape 2.2 l'opérateur va attribuer l'incident à l'hopital adéquat ;
  - étape 2.3 l'opérateur identifie l'incident comme traité (traitement à "true") ;
  - étape 2.4 l'opérateur se remet au statut "Disponible" ;
  - étape 2.5 boucle sur l'étape 2.1.
- Etapes 3 de finalisation (actions des hôpitaux) :
  - étape 3.1 l'hôpital prend en charge l'incident.

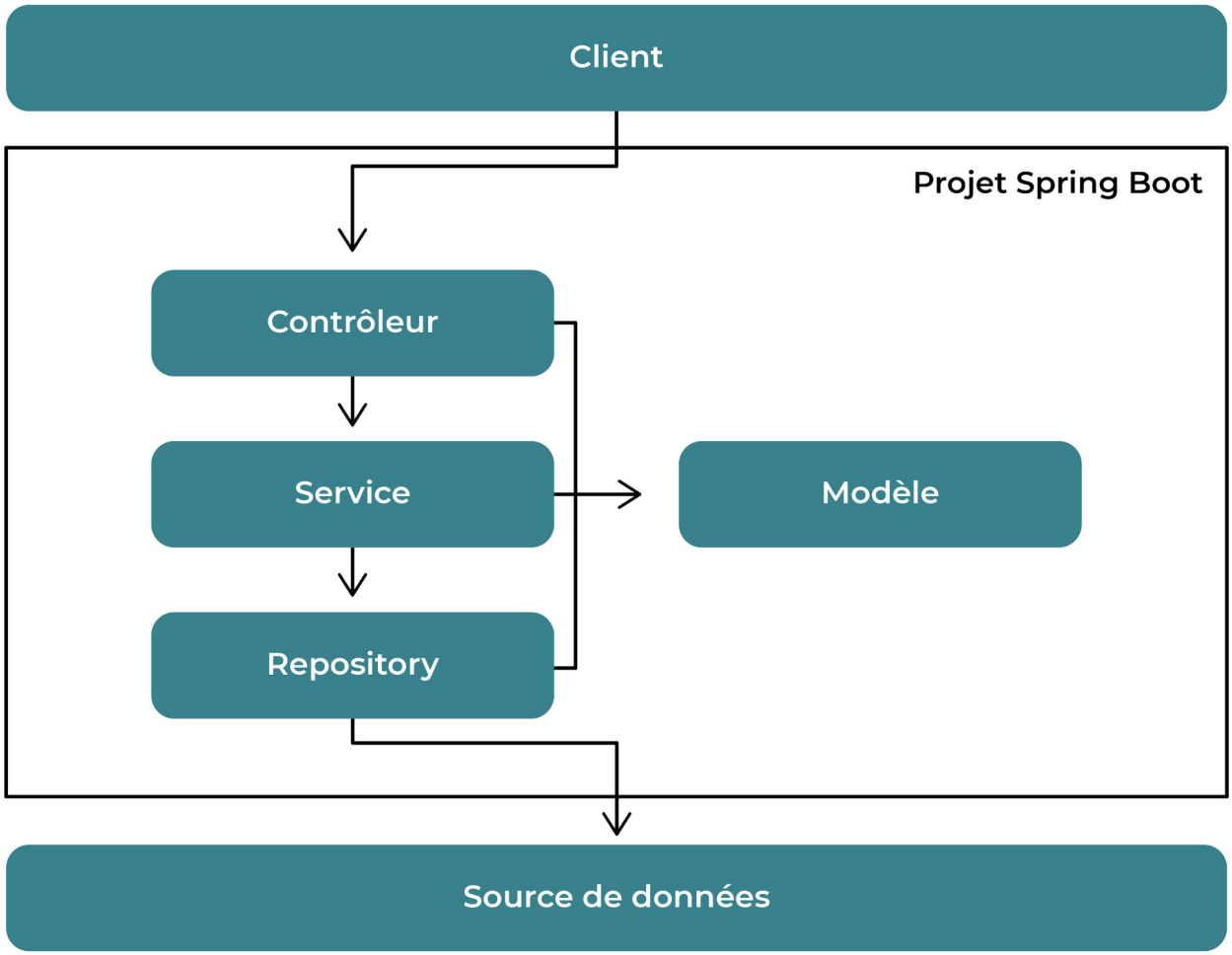
Les différentes étapes recensés ci-dessus pourront alors se représenter graphiquement comme suit :



III. Fonctionnalités spécifiques et attributs

III.A. Structure commune

Cette PoC va être réalisé à l'aide du projet Spring Boot et sa structure suivra donc les préconisations de Spring Boot, à savoir :



En outre, comme il a été présenté dans les paragraphes précédents, il est possible de recenser sept composants au total pour cette PoC.

Techniquement, pour bien mettre en évidence la MSA et l'indépendance de chaque composant logiciel, ils seront tous architecturés comme suit :

```
<composant>
… src
… … main
… … … java
… … … … org.medhead.emergencySystem.<composant>
… … … … … controller
… … … … … model
… … … … … repository
… … … … … service
… … … … … api<Composant>Application.java
… … … … … CustomProperties.java
… … … resources
… … … … templates
… … … … application.properties
```

Dans la structure ci-dessus, réalisée pour mettre en avant une topologie basée » sur Spring Boot, il sera nécessaire de porter attention aux répertoires suivants :

- `<composant>/src/main/org.medhead.emergencySystem.<composant>/controller` : ce répertoire représente la couche Controller chargée des intéractions entre l'utilisateur de l'application et l'application elle-même. Ainsi, elle contiendra les classes de publication des *endpoints* de l'application associés à chaque URL pour pouvoir communiquer en HTTP.
- `<composant>/src/main/org.medhead.emergencySystem.<composant>/model`: ce répertoire représente la couche Service implémentant des traitements métiers spécifiques à l'application. Ainsi, elle contiendra les entités représentant la table de base de données dudit composant.
- `<composant>/src/main/org.medhead.emergencySystem.<composant>/repository`: ce répertoire représente la couche Repository intéragissant avec les sources de données externes. Ainsi, elle contiendra les interfaces permettant d'éxécuter les requêtes SQL à partir de *Spring Data JPA*.
- `<composant>/src/main/org.medhead.emergencySystem.<composant>/service`: ce répertoire représente la couche implémentatnt des objets métiers qui seront manipuliés par les autres couches. Ainsi, elle contiendra les traitement dictées par les règles fonctionnels et de gestion de l'application.

III.B. Structure de chaque composant

III.B.1. Couche model

La couche model va recenser tous les attributs de chaque composant. Les attributs correspondront alors au nom des colonnes des tables de la base de données afférente.

Les annotations Spring Boot nécesssaires à cette couche sont `@Data`, `@Entity` et `@Table(name=« table »)`.

III.B.2. Couche repository

Cette couche va représenter les opérations CRUD types. Il est alors à noter ici que Spring Boot prend en charge automatiquement la gestion de ces opérations en important le package org.springframework.data.repository.CrudRepository.

L'annotation Spring Boot nécesssaire à cette couche est `@Repository`.

III.B.3. Couche Service

Cette couche va prendre à sa charge les traitements métiers nécessaires pour la PoC. Par exemple, en ce qui concerne le composant `apiIncidents`, il sera alors possible d'y déclarer les méthodes suivantes :

- `Optional<Incident> getIncident(Long id)` : méthode retournant un objet Incident en fournissant son id en paramètre.
- `Iterable<Incident> getIncidents()` : méthode retournant la liste de tous les incidents déclarés.
- `void deleteIncident(Long id)` : méthode supprimant un incident en fournissant un son id en paramètre.
- `Incident saveIncident(Incident incident)` : méthode sauvegardant un objet Incident passé en paramètre au sein de la base de données des incidents.

Les annotations Spring Boot nécesssaires à cette couche sont `@Data` et `@Service`.

III.B.4. Couche controller

Cette couche va interpréter et traiter les requêtes HTTP passées au client web et les transmettre à l'obet métier en cours. Les principales méthodes qui seront utilisées seront :

```
@PostMapping(« /composant »)
Composant createComposant(@RequestBody Composant composant() {...}
```

```
@GetMapping(« /composant/{id} »)
Composant getComposant(@PathVariable(« id ») Long id) {…}
```

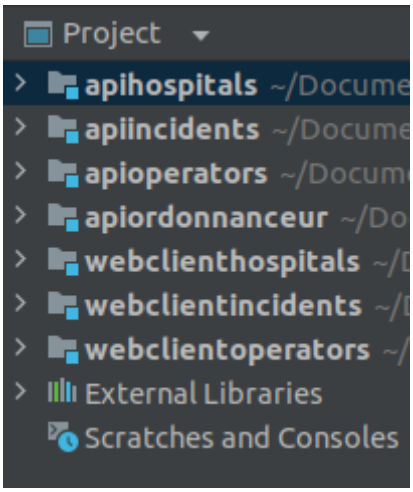
```
@GetMapping(« /composants»)
Iterable<Composant> getComposants() {…}
```

```
@PutMapping(« /composant/{id} »)
Composant updateComposant(@PathVariable(« id ») Long id,
@RequestBody Composant composant) {…}
```

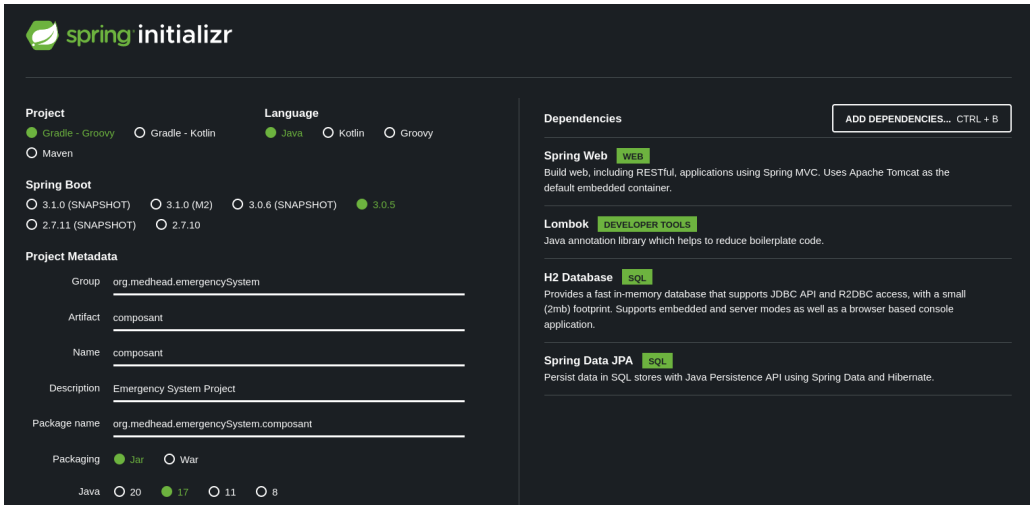
```
@DeleteMapping(« /composant/{id})
void deleteComposant(@PathVariable(« id » Long id) {…}
```

III.B.5. Composants

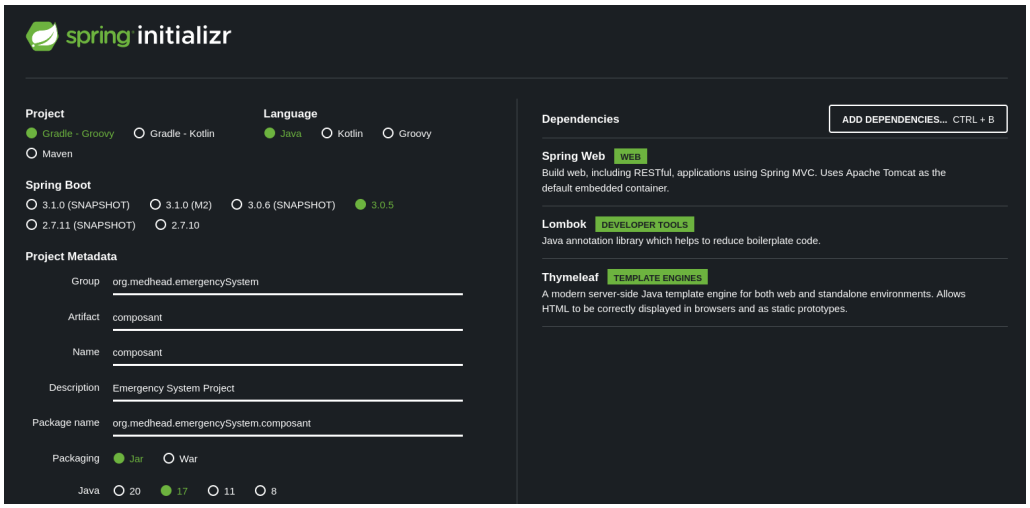
L’ensemble des sept composants peuvent être regroupés au sein du même projet au sein de l’IDE :



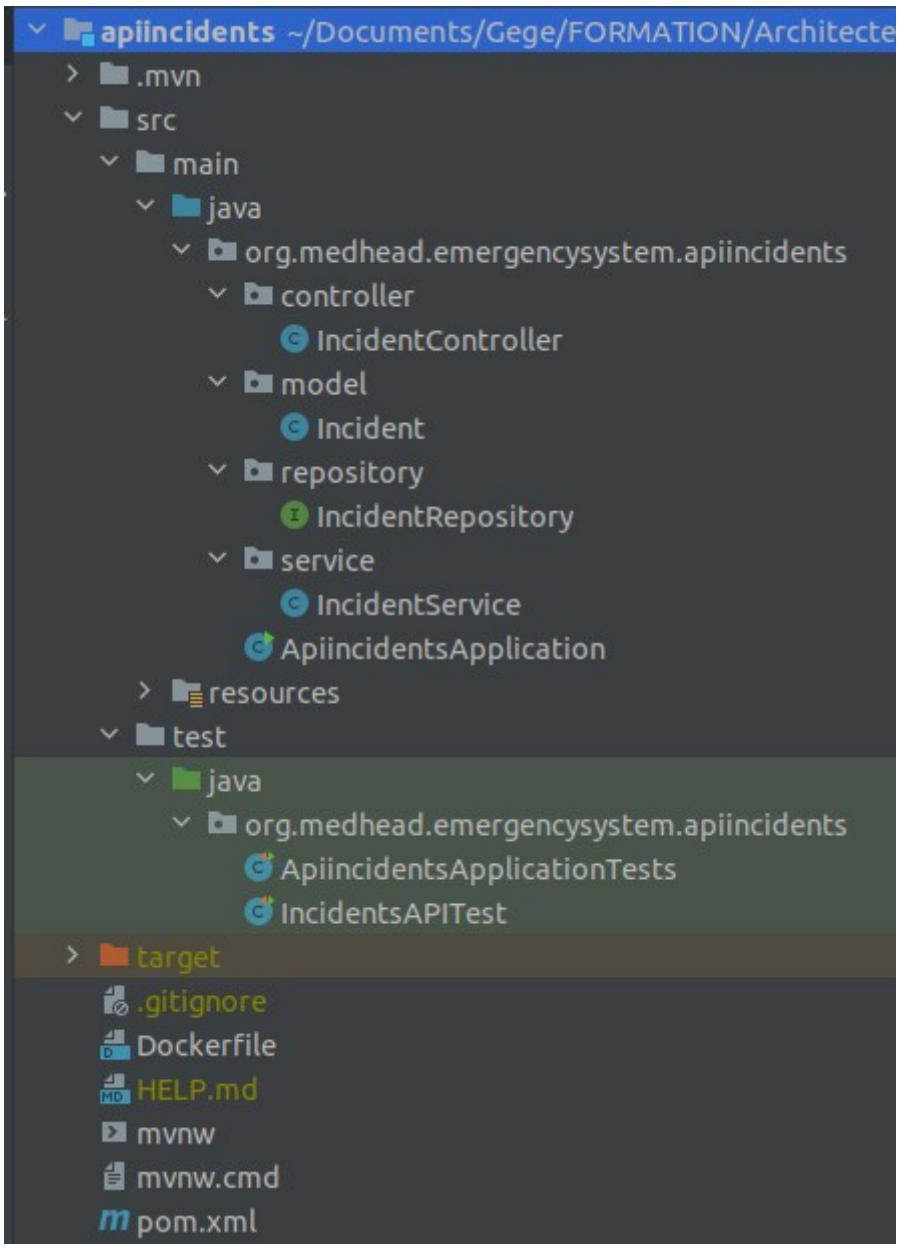
Les API auront alors une base Spring Boot sous Spring Initializr :



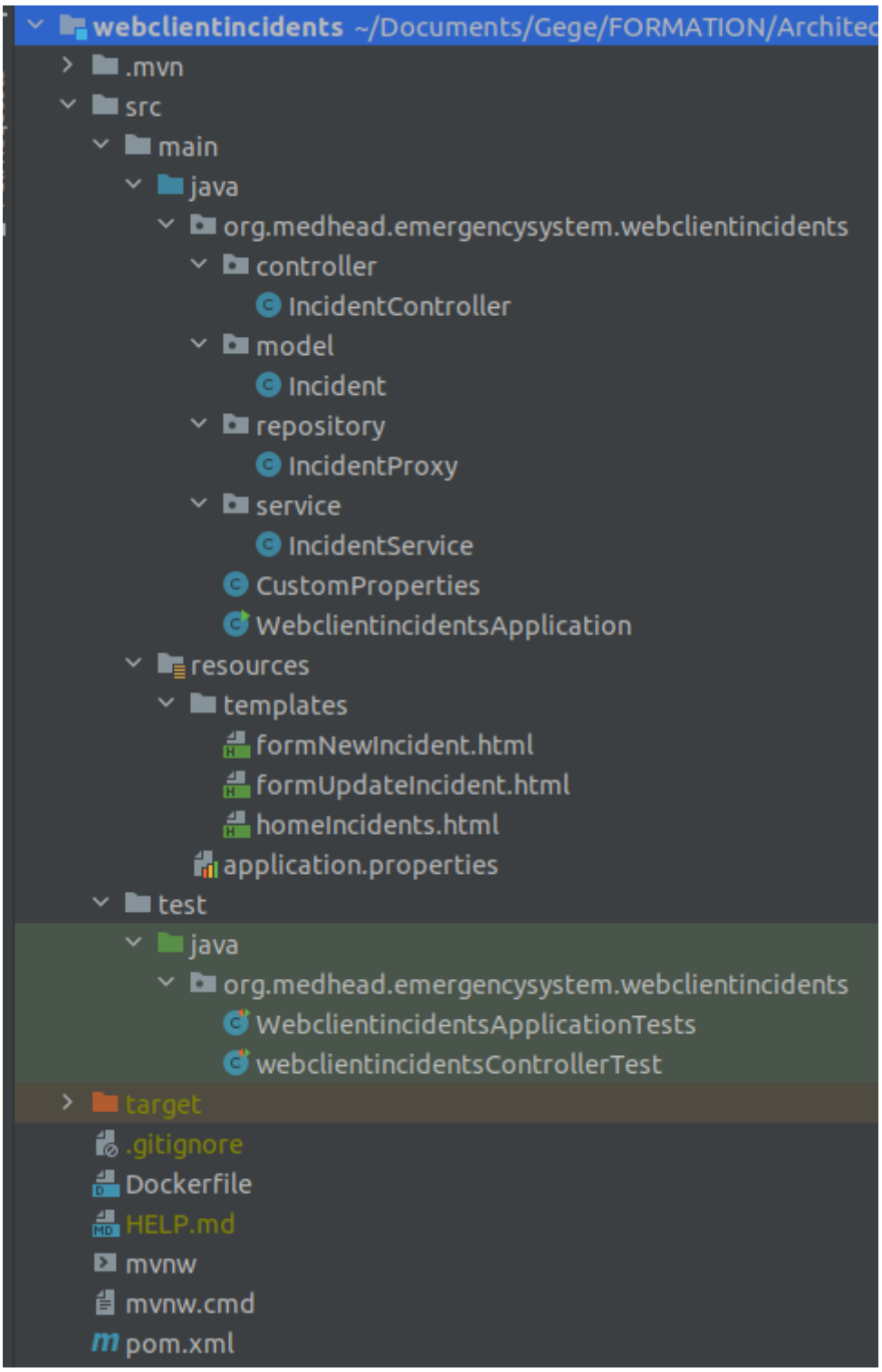
Les clients web auront une Spring Boot légèrement différente :



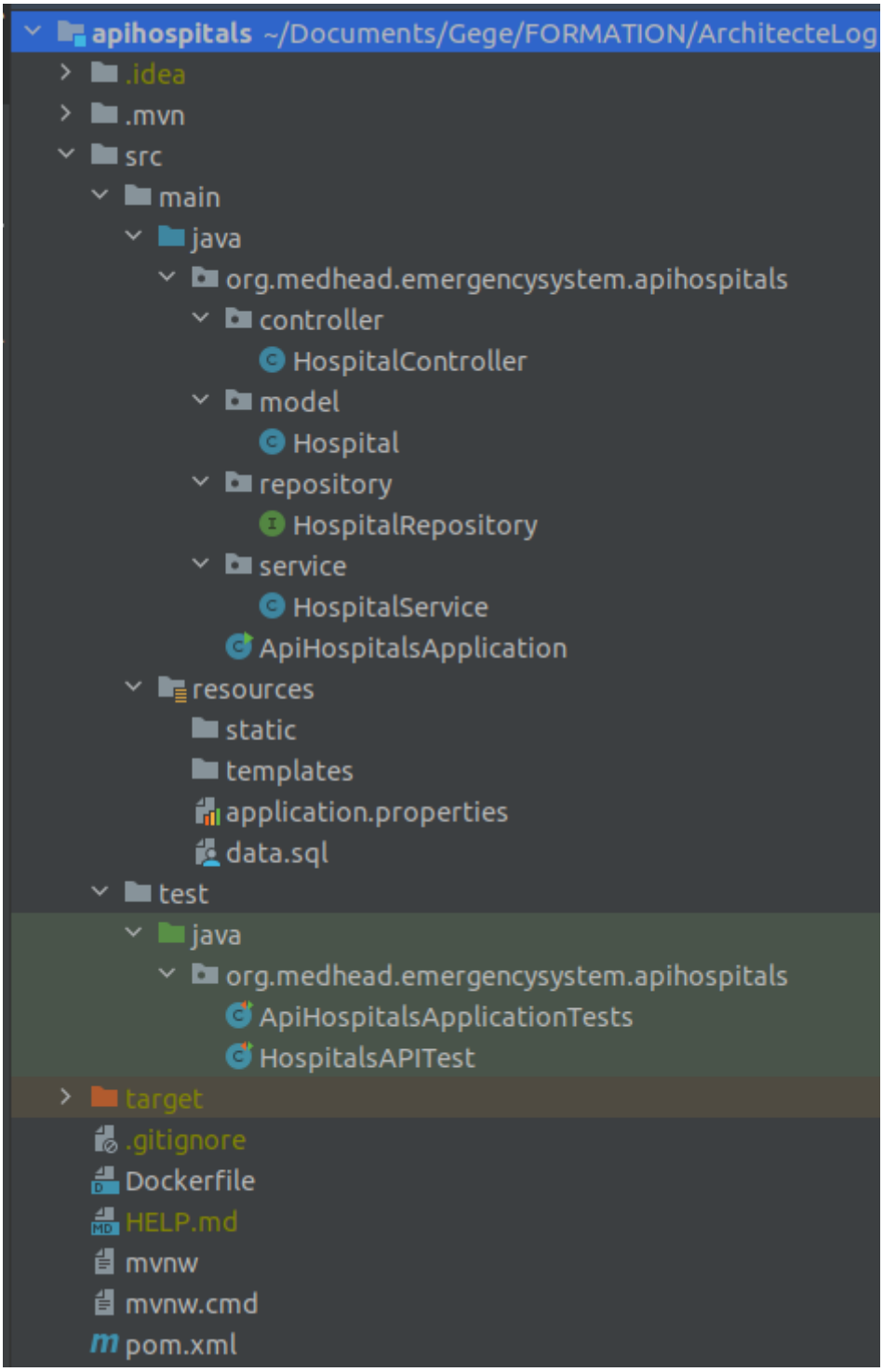
III.B.5.a. Composant apiIncidents



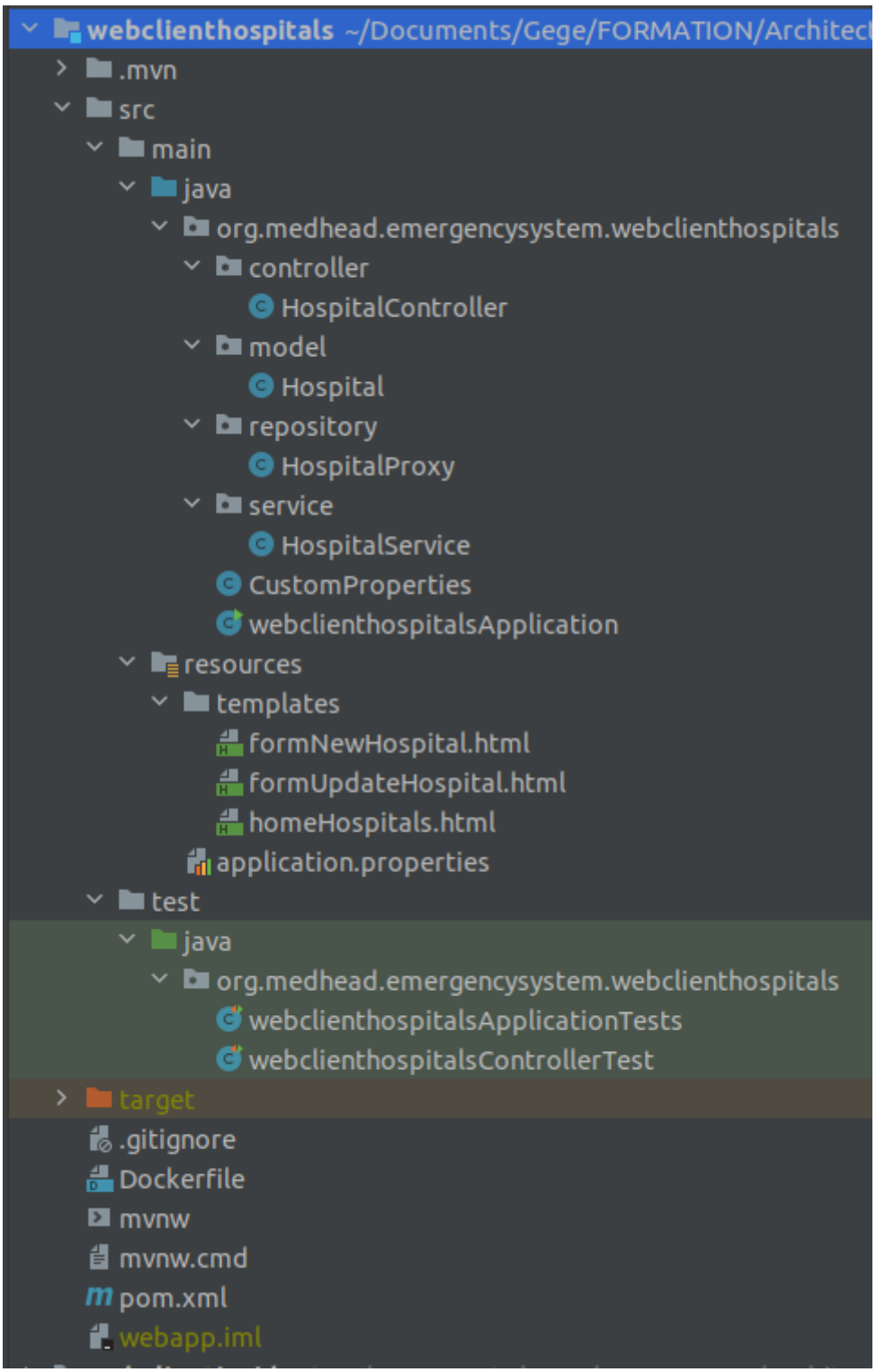
III.B.5.b. Composant webclientIncidents



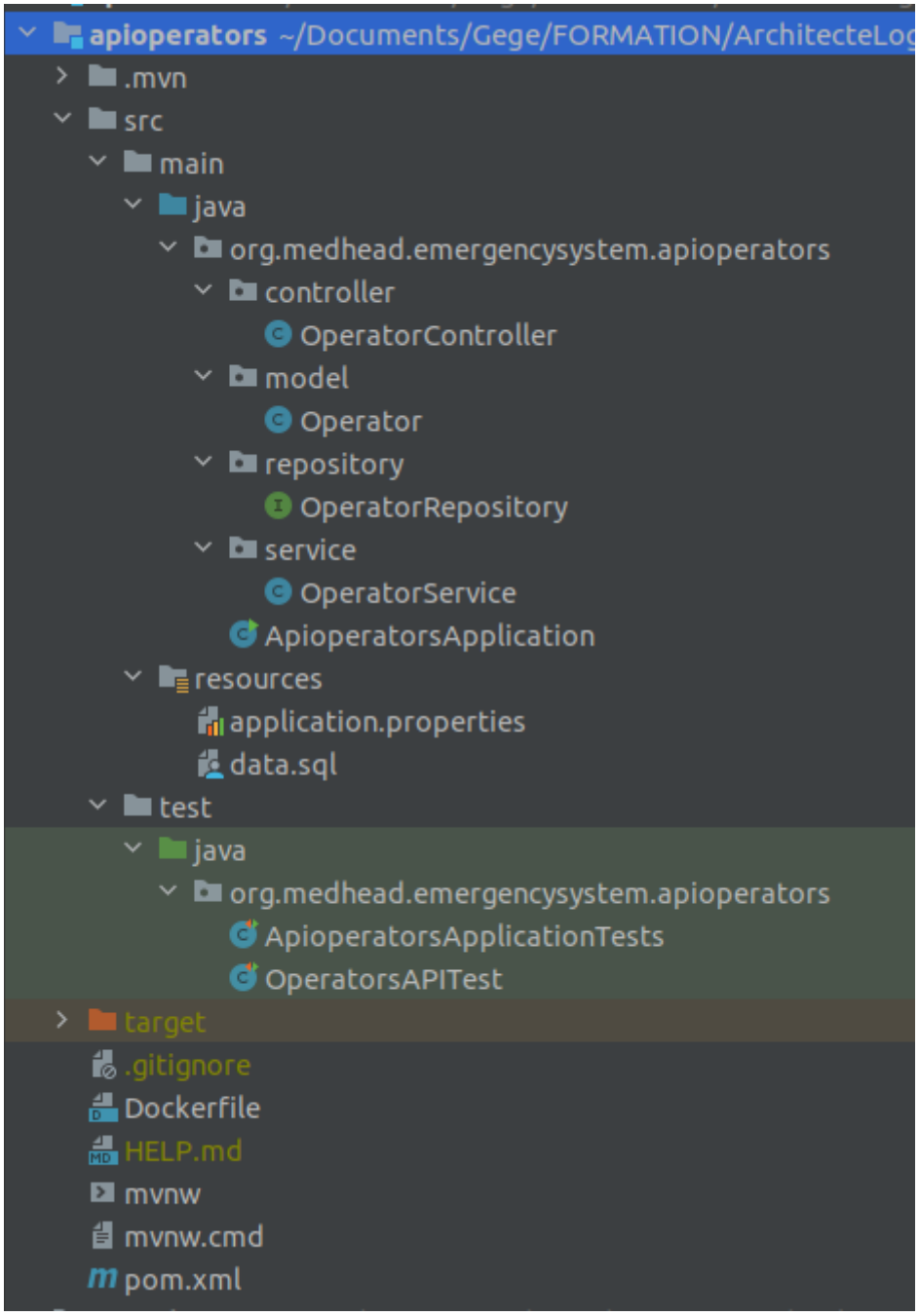
III.B.5.c. Composant apiHospitals



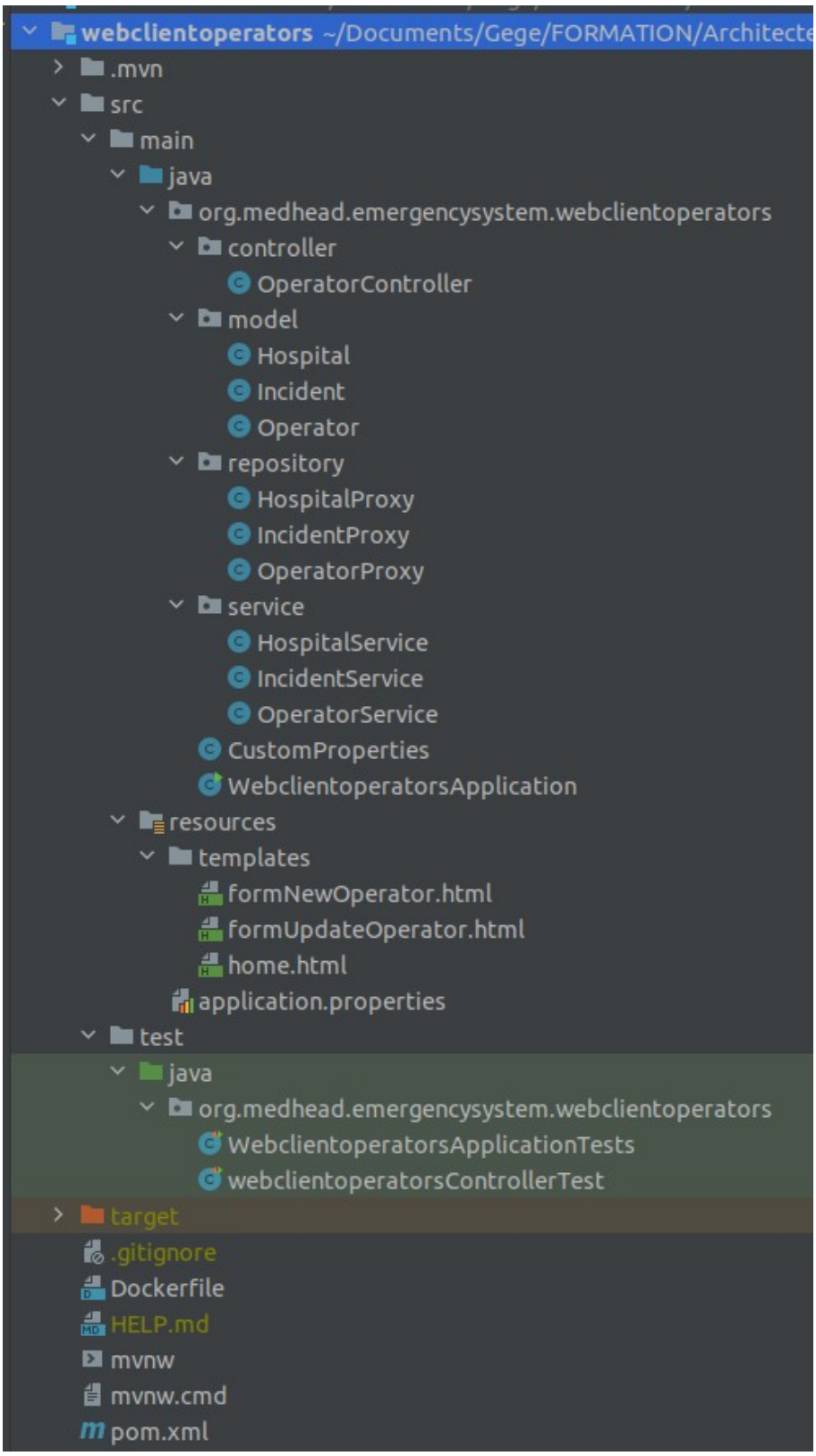
III.B.5.d. Composant webclientHospitals



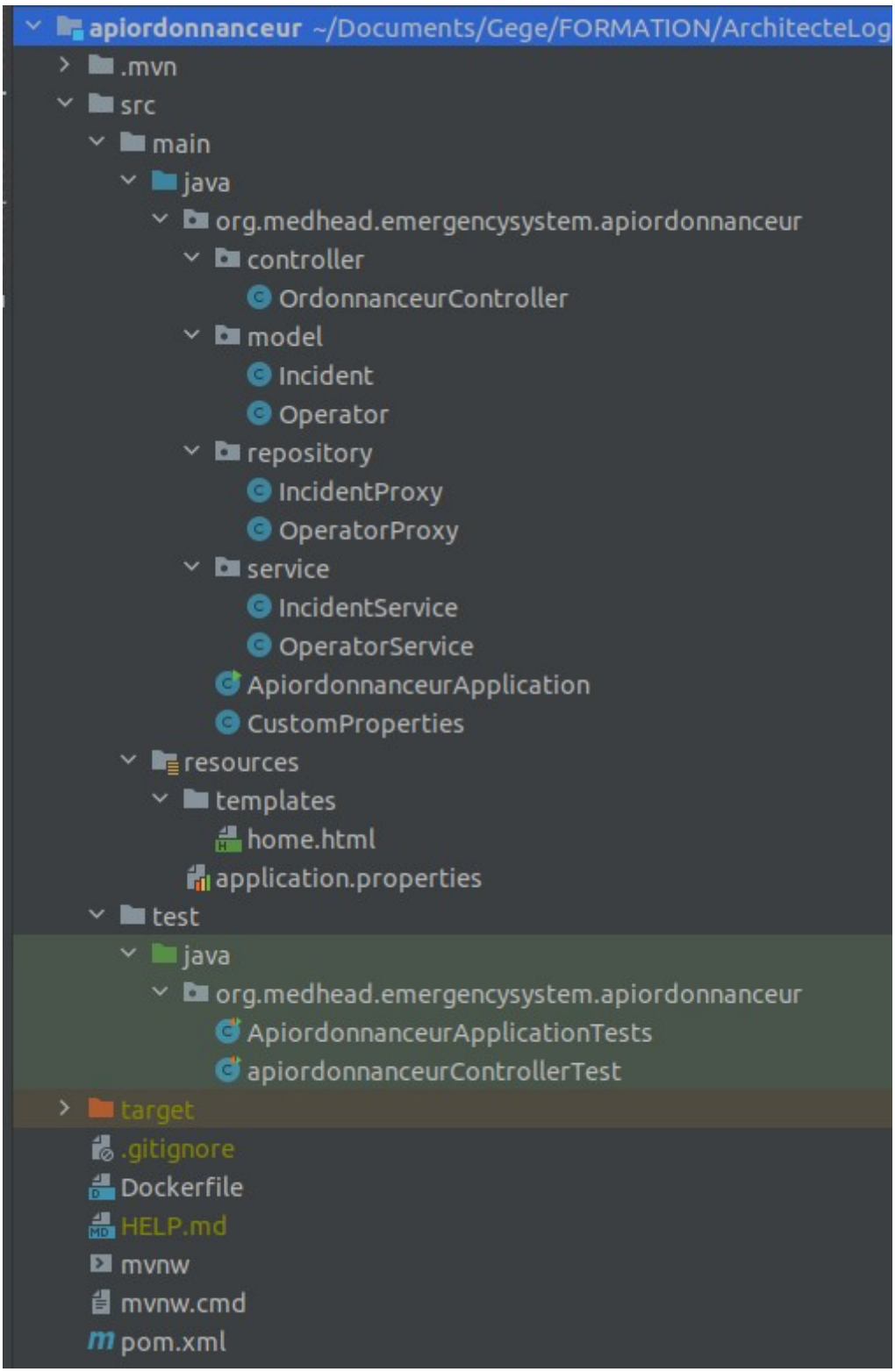
III.B.5.e. Composant apiOperators



III.B.5.f. Composant webclientOperators



III.B.5.g. Composant apiOrdonnancer




MedHead